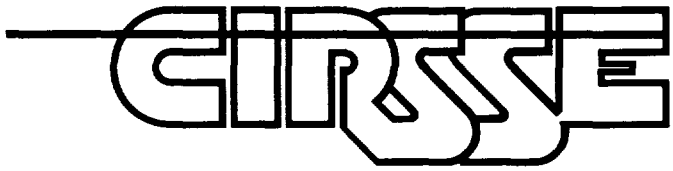


Center for Intelligent Robotic Systems for Space Exploration

Rensselaer Polytechnic Institute
Troy, New York 12180-3590

EFFICIENT IMPLEMENTATION OF LINEAR AND QUADRATIC PROGRAMMING ALGORITHMS FOR MINIMUM DISTANCE ESTIMATION BETWEEN SOLIDS

By:

J. Whitehead
K.J. Kyriakopoulos

Department of Electrical, Computer and Systems Engineering
Department of Mechanical Engineering, Aeronautical Engineering & Mechanics
Rensselaer Polytechnic Institute
Troy, New York 12180-3590

May 1989

CIRSSE Document #22

PROJECT OVERVIEW

The following document is the final report and project documentation for work accomplished by Jim Whitehead [redacted] to meet the requirements of a Senior Project. (Course 35498, Spring '89 Semester, Rensselaer Polytechnic Institute.) The objective of this project was to write computer code to perform Linear Programming and Quadratic Programming minimization tasks, for eventual inclusion in a robotic control system. The basic concept underlying the project is this: given a description of a robot and a description of an object in the robot's surrounding in terms of a series of inequalities, the minimization code can find the minimum distance between the robot and the object. The linear programming algorithm minimizes the infinity norm, while the quadratic programming algorithm minimizes the euclidian norm.

The programming language "C" was used to implement both programming algorithms, due to its natural speed, combined with the ability to use different floating point libraries at will. An added advantage of using "C" on the Sun workstations is the ability to make use of the 68881 floating point coprocessor on the Sun 3 computers, which provides dramatic speed increases over using floating point libraries. Each programming algorithm was implemented as a function, along with its supporting subroutines. For ease of development, each algorithm was developed in a separate program. However, there should be no difficulty in incorporating either the linear programming algorithm or the quadratic programming algorithm into a larger section of code when implementing a robotic control system.

The linear programming algorithm implements the two phase simplex method, originally attributable to George Dantzig. However, unlike Dantzig's tableau form of the simplex algorithm, the method implemented in the linear programming code has been optimized for machine computation. Many steps have been taken to increase the speed of the simplex algorithm, with the final result being extremely quick code. Using one set of test data, a Sun 3 computer achieved an average speed of 33.3 simplex method

minimizations per second (5 variables, 7 constraints). The linear programming algorithm was also designed to allow for any type of constraint to be used, thus allowing for greater flexibility in problem definition.

The quadratic programming algorithm implements the Gradient Projection Method for the Quadratic Programming case. (i.e., it implements the Gradient Projection Method with an objective function $f(\mathbf{x}) = \mathbf{x}'\mathbf{Q}\mathbf{x}$). This quadratic programming algorithm has been modified for more proper use in a robot control setting, and thus is not as general as it might be. It assumes an initial knowledge of a feasible point, information that a vision system can provide. It also assumes that this initial point will be fairly close to optimal, thus ensuring a low number of quadratic programming iterations to find the optimal position. Hence speed was not a primary objective in this code. While the coding of the quadratic programming algorithm took advantage of many ways to increase the computational speed, a greater emphasis was placed on providing code that was easy to maintain. Thus the main design objective was to provide easily understood and well-documented code. Since there is the possibility that this code may be used in a limited memory situation, a second version of this algorithm exists which sacrifices clarity of code for memory efficiency by using pointer arithmetic instead of array subscripting. This ensures that there is no wasted array space in the algorithm. To help modifications of the code, a debug flag exists which, if set, will cause the program to print the state of the algorithm during each iteration.

Each programming algorithm will now be discussed more fully. Descriptions of all functions, a block diagram of the algorithm, a trial run, program listings, and files usage will be given for each algorithm.

Linear Programming Algorithm

Simplex Algorithm

The linear programming algorithm has been implemented by using a two phase simplex algorithm approach. Though based on theoretical work by George Dantzig, the method and formulations used are taken from the book Linear Programming: An Introduction to Finite Improvement Algorithms, by Daniel Solow.

Linear programming is a form of optimization where a linear function is optimized subject to a series of linear constraints. In mathematical form, a linear programming problem can be formulated as follows:

minimize **Cx**

subject to **Ax** = **b** ; **x** >= 0

In this representation, **C** is a vector representing the "cost" of each variable, **A** is a matrix of constraints (which can contain greater than or equal to, less than or equal to, or equal to type constraints), **b** is a matrix of constraining values, and **x** is the variable to be optimized. Given a linear programming problem expressed in this form, a function in the "C" program called Simplex will implement a two phase simplex algorithm to find the minimum value of **Cx** subject to **Ax** = **b**.

The simplex method operates by separating the columns of the constraint matrix **A** into two sets -- a basis set, and non-basis set. If a column of **A** is in the basis, its associated element in **x** is considered to be basic. The value of **x**, when it contains the values of its basic variables, and the value zero for non-basic variables is called a basic feasible solution (if **A** is m x n, there are exactly m basic variables). A basic feasible solution is a point on the boundary of the constraint space defined by **A**. Unfortunately, there is no easy way to determine a basic feasible solution of the system of equations from scratch, and thus the Phase I procedure must be used.

Phase I finds an initial basic feasible solution for the linear programming problem. Phase I accomplishes this by augmenting the **A** matrix with columns from the identity matrix to form an initial basis consisting of the identity matrix. Since each additional column has a new variable associated with it, the colection of additional variables are called artificial variables. Phase I then calls the simplex algorithm with a new cost function to remove the artificial variables from the basis. When all the artificial variables has been removed from the basis, the current basis is a basic feasible solution of the constraint equation **Ax** = **b**. The Phase I procedure operates independently of the original cost function (**Cx** in the linear case), and thus could be used to find a basic feasible solution for a quadratic programming solver too.

Once Phase I has been completed, Phase II begins. Phase II takes the current basis and basic feasible solution, along with the original cost function, **C**, and calls the Simplex algorithm. The Simplex algorithm then proceeds to quickly solve the original problem, since the Phase I solution is usually quite near to the optimum point. Phase II also takes care of the case where some artificial variables might have the value 0 at the end of Phase I, but with the associated column still in the basis.

The simplex algorithm, called by both Phase I and Phase II, is an iterative procedure used to minimize a linear function, **Cx**, subject to linear constraints, **Ax** = **b**. The simplex algorithm begins with an initial feasible solution, **x**, the matrix of the columns of the basis, **B**, and the matrix of the columns not in the basis, **N**. During each iteration of the simplex algorithm, seven operations are performed, these being a test for optimality (are we done yet?), calculation of a direction of increasing optimality (if we aren't done, which direction should we move in to get closer to being done?), selection of a column to leave the basis (by moving in the direction of increasing optimality, you start at one point on the simplex...), selection of a column to enter the basis (... and end at another point on the simplex), calculation of an amount to move in the direction of increasing optimality, recalculation of the basis inverse matrix, and updating the position vector.

The test for optimality is performed by calculating the reduced costs vector by using the following equation:

$$\textbf{selbas} = C_N - C_B B^{-1} N$$

If the **selbas** vector is greater than or equal to zero, the linear program is optimal. Otherwise, the simplex algorithm picks the most negative element of **Selbas** and stores the value of the column in a variable j^*, which holds the value of the column to enter the basis.

Next, the direction of optimality is calculated from the following equation:

$$d_B = -B^{-1} * N_{.j^*}$$

If d_B is greater than or equal to zero, the linear program is unbounded, and will return an error code. Otherwise, the simplex algorithm calculates the minimum of all the: $- x_B / d_B$ for d_B less than 0. The column of the basis corresponding to this minimum is called k^* and is the column that will leave the basis. The actual value of this minimum is the amount to be traveled in the direction of optimality, denoted t^*.

Next, the simplex algorithm replaces column k^* of **B** with column j^* of **N**, thus creating a new basis. This requires the updating of the B^{-1} matrix. This can be accomplished by performing the following update equation:

$$\left(B'^{-1}\right) = \begin{cases} \left(B^{-1}\right)_{i\cdot} - \left[(d_B)_i / (d_B)_{k^*}\right]\left(B^{-1}\right)_{k^*\cdot} & \text{if } i \neq k^* \\ -\left[1/(d_B)_{k^*}\right]\left(B^{-1}\right)_{k^*\cdot} & \text{if } i = k \end{cases}$$

At this point, the simplex routine calculates the new value of x, by finding: $\mathbf{x}' = \mathbf{x} + t^*\mathbf{d}$. The simplex algorithm then loops back to the test for optimality.

Thus concludes a brief overview of the two phase simplex method implemented in this project. The object code, in "C", can be found in the file **lp.c**. The simplex routine is easily incorporated into other code by following these steps: copy **lp.c** into the new object file; remove all but the variable declarations and Simplex function call from the **lp.c** main() routine; and copy the three #define's into your program's #define block.

Following this description of the simplex algorithm, a brief description of every function in the program **lp.c** will be given, followed by a block diagram, a trial run, and the listing of **lp.c**.

The simplex algorithm was performed 200 times on the problem shown in the trial run, and the speed of the algorithm was timed, with the following result. The total run took 6 seconds. Each minimization required four simplex iterations, for a grand total of 800 simplex iterations. This works out to 133 iterations/second, and 33.3 minimizations/second. This case had 5 variables, and 7 constraint equations. For a real-time application, the linear programming algorithm should be feasible for up to 20 or so objects in a robot's surroundings, assuming distance updates every second.

FUNCTION DESCRIPTIONS

The following is a listing and brief description of every function in the lp.c code. If more information about these routines is desired, please consult the well-documented source code.

LUFact

The LUFact routine performs an LU factorization on an input array. The basic concept is that, in cases where gaussian elimination is to be repeatedly performed on a block matrix (such as during the calculation of a matrix inverse), it is easier to separate the steps of gaussian elimination and backwards solution. This spares the computational expense of repeated gaussian eliminations on the same matrix (the end result will be the same each time). An LU factorization performs this gaussian elimination step by finding the two matricies **L** & **U** such that a block matrix **A** (as in **Ax** = **b**) can be represented as **LU**. The LU factorization is stored in the original matrix. This is accomplished by assuming the diagonal elements of **L** are all 1.

The method of LU factorization used includes pivoting. Instead of swapping elements in **A** to accomplish this swapping operation, indicies in a pivot matrix are swapped. Thus, in addition to **A**, this routine needs a pivot matrix (internally called piv) initially set to equal [0, 1, 2, ... blksize]. The LUFact routine will change these values if pivoting is necessary.

The actual method implemented comes from pages 345-346 of Numerical Analysis, Third Edition by Richard Burden and Douglas Faires, and references to step numbers in comments in the code refer back to this source.

Input:

array - an array of size blksize x blksize on which to perform LU factorization
piv - a vector of pivot values, as described above. Input values should be [0, 1, 2, ... blksize].
blksize - the size of the block matrix array

Output:

array - in LU factored form. Assumes diagonals of L are 1.
piv - reflects row pivoting of array performed during LU factorization.

LUSolv

Once an array is in the form **A** = **LU**, the system **Ax** = **b** can be solved by finding **z** such that **Lz** = **b**, and then by finding **x**, such that **Ux** = **z**. This routine performs this backwards solution routine.

Input:

Mat - the output of LUFact on a matrix
piv - the result of pivot ooperations in LUFact
blksize - the size of A
b - the **b** in **Ax** = **b**

Output:

x - the **x** in **Ax** = **b**, solved.

Invert

A matrix inverse is found by repeatedly solving the system **Ax** = **b** for **b** equal to columns from the identity matrix, and with **x** returning the appropriate column in the matrix inverse. Since this requires the repetitive solving of systems involving **A**, this matrix is LU decomposed before repeatedly calling LUSolv. This makes this a very speedy implementation of the matrix inverse function.

Input:

Mat - the array to be inverted
blksize - the size of Mat is blksize x blksize

Output:

Matinv - the matrix inverse of Mat

PrimSimp

This routine is the heart of the whole program -- it implements the simplex algorithm, explained in detail previously. While normally this routine would be called with both the current **x** vector and B^{-1} matricies known, there are two flags, Bflag and Xflag, which can be set if a matrix inverse or initial **x** calculation is desired. The basis set and non-basis set are contained in two vector arrays. If columns 0, 2 & 4 were in the basis, the vector array inbas would contain the elements [0, 2, 4], with the non-basis vector array outbas containing [1,3,5,...]. An error code can be sent back from the PrimSimp routine in the case that the linear programming problem is unbounded. This occurence is noted by an *err value of 2, and should be checked for in the calling program.

Each step of the simplex algorithm is noted by comments in the code -- even though the code is not broken up into many subroutines, it should still be fairly easy to follow program flow.

Input:

A - constraint matrix
b - constraint values matric
C - costs vector
x - an initial basic feasible solution (if Xflag is equal to 1)
Binv - the matrix B^{-1} (if Bflag is equal to 1)
inbas - the set of current columns of the basis
outbas - the set of columns not in the basis
M,N - A is M x N, b is M x 1, x is N x 1
Bflag - set to 0 if Binv is not already known

Xflag - set to 0 if **x** is not already known

Output:

x - value of the variables at the minimum value of the cost function
*mc - the value of the cost function at minimum
*iter - the number of iterations performed in reaching the minimum

ConvertToStandard

Adds slack variables, as needed, to the matrix **A**, for greater than or equal to or less than or equal to type constraints. This simplex algorithm assumes **A** to contain constraints of the form **Ax** = **b**, and this routine ensures compliance to that format.

Input:

A - a matrix of constraint equations
eq - the first eq rows of A represent equality constraints
lt - the next lt rows of A represent less than or equal to type constraints
gt - the last gt rows of A represent greater than or equal to type constraints
M,N - A is M x N.

Output:

A - (modified) For the first eq rows, no change. For the next lt rows, A has a positive slack variable added to represent the difference between the amount less than and the amount equal to. For the last gt rows, A has a negative slack variable added to represent the difference between the amount greater than and the amount equal to.

Simplex

This routine performs the Phase I and Phase II steps of the linear programming algorithm. This routine is very general. It can perform a minimization or a maximization, and accepts a constraint matrix with equations of all three constraint types inside. This routine returns an error code in *err. If *err = 2, then the linear program is unbounded. If *err = 4, then the linear program is infeasible. The calling program should check for these conditions.

This routine, rather than PrimSimp is the front-end of the linear programming algorithm.

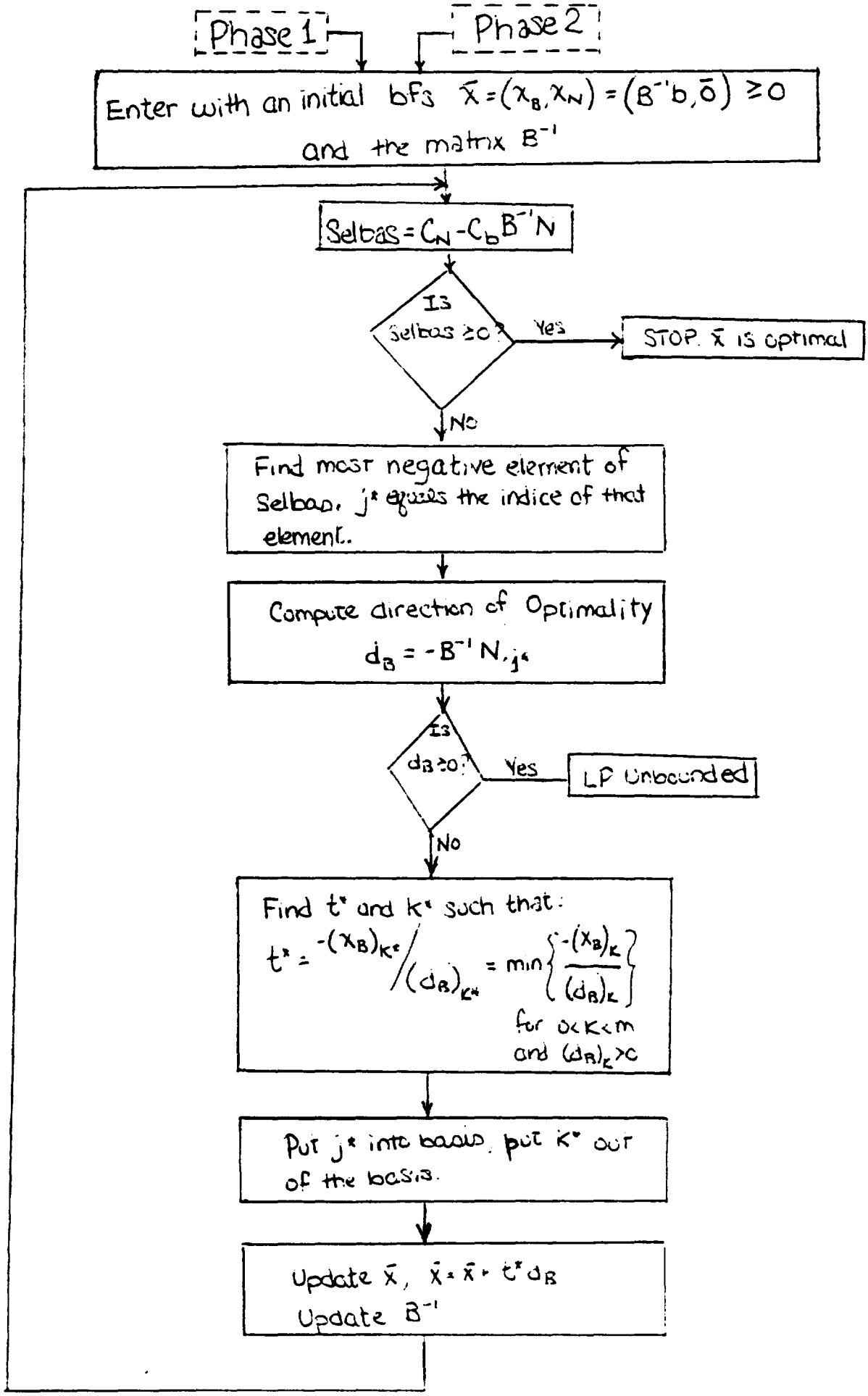
Input:

A - a matrix of constraint equations
b - constraining values vector
C - coeffients of the cost function
M,N - matrix sizes
Mflag - set to 0 to minimize, set to 1 to maximize
eq - the first eq rows of A are equality constraints
lt - the next lt rows of A are less than or equal to constraints
gt - the last gt rows of A are greater than or equal to constraints

Output:

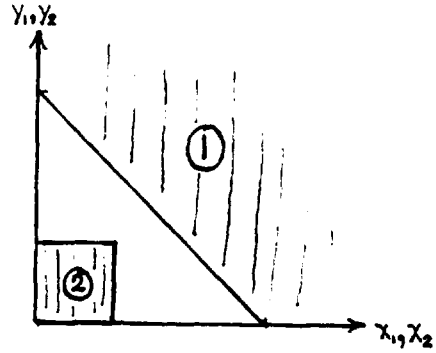
x - the value of the variables at the optimum point
*mc - the value of the cost function at optimum
*err - a 2 if LP unbounded, a 4 if LP infeasible

BLOCK DIAGRAM OF THE LINEAR PROGRAMMING ALGORITHM


Phase 1
Phase 2
Enter with an initial bfs $\bar{x} = (x_B, x_N) = (B^{-1}b, \bar{0}) \geq 0$ and the matrix B^{-1}
$Selbas = C_N - C_B B^{-1} N$
Is Selbas ≥ 0?
Yes
STOP. $\bar{x}$ is optimal
No
Find most negative element of Selbas. j^* equals the indice of that element.
Compute direction of Optimality
$d_B = -B^{-1} N_{,j^*}$
Is $d_B \geq 0$?
Yes
LP Unbounded
No
Find t^* and k^* such that:
$t^* = -(x_B)_{k^*} / (d_B)_{k^*} = \min \left\{ \frac{-(x_B)_k}{(d_B)_k} \right\}$
for $0 < K < m$ and $(d_B)_k > 0$
Put j^* into basis, put k^* out of the basis.
Update $\bar{x}$, $\bar{x} = \bar{x} + t^* d_B$
Update B^{-1}

TRIAL RUN

The following pages contain a trial run of the linear program solver program. The physical system being solved is as follows:



This physical system is represented by the following system of equations:

$$\underset{\bar{A}}{\begin{bmatrix} -1 & -1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & -1 & 0 & -1 \\ -1 & 0 & 1 & 0 & -1 \\ 0 & 1 & 0 & -1 & -1 \\ 0 & -1 & 0 & 1 & -1 \end{bmatrix}} \underset{\bar{x}}{\begin{bmatrix} x_1 \\ y_1 \\ x_2 \\ y_2 \\ w \end{bmatrix}} \leq \underset{\bar{b}}{\begin{bmatrix} -3 \\ 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}}$$

$$\text{minimize } \bar{c}\bar{x}$$

$$\bar{c} = \begin{bmatrix} 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

```
venus% lp


Linear Program Solver
 (Simplex Method)


Please enter the number of variables: 5

Please enter the number of constraint equations: 7


Constraint equations can be one of three types, either
equality constraints, less than constraints, or greater than
constraints. Please enter three numbers below to specify how
many of each type equation are present in your constraint
equation matrix.

Please enter how many equality constraints are present: 0

Please enter how many less than or equal to type
constraints are present: 7

Please enter how many greater than or equals to type
constraints are present: 0



Please enter a ten character (or less) description of each variable.

Variable  1: X1

Variable  2: Y1

Variable  3: X2

Variable  4: Y2

Variable  5: W


For each constraint equation, please enter all variable coefficients below:


Less than or equal to Constraints:

Equation # 1:

Coefficient of X1 : -1
Coefficient of Y1 : -1
Coefficient of X2 : 0
Coefficient of Y2 : 0
Coefficient of W : 0

Constraining value: -3
```

Equation # 2:

Coefficient of X1 : 0
Coefficient of Y1 : 0
Coefficient of X2 : 1
Coefficient of Y2 : 0
Coefficient of W : 0

Constraining value: 1

Equation # 3:

Coefficient of X1 : 0
Coefficient of Y1 : 0
Coefficient of X2 : 0
Coefficient of Y2 : 1
Coefficient of W : 0

Constraining value: 1

Equation # 4:

Coefficient of X1 : 1
Coefficient of Y1 : 0
Coefficient of X2 : -1
Coefficient of Y2 : 0
Coefficient of W : -1

Constraining value: 0

Equation # 5:

Coefficient of X1 : -1
Coefficient of Y1 : 0
Coefficient of X2 : 1
Coefficient of Y2 : 0
Coefficient of W : -1

Constraining value: 0

Equation # 6:

Coefficient of X1 : 0
Coefficient of Y1 : 1
Coefficient of X2 : 0
Coefficient of Y2 : -1
Coefficient of W : -1

Constraining value: 0

Equation # 7:

Coefficient of X1 : 0
Coefficient of Y1 : -1
Coefficient of X2 : 0
Coefficient of Y2 : 1
Coefficient of W : -1

Constraining value: 0

```
For each variable, please enter an associated cost to minimize.
Please enter the cost for X1 : 0
Please enter the cost for Y1 : 0
Please enter the cost for X2 : 0
Please enter the cost for Y2 : 0
Please enter the cost for W : 1
Simplex: LP Optimized.
Final variable values:

X1 :     1.50000
Y1 :     1.50000
X2 :     1.00000
Y2 :     1.00000
W :    0.50000
Minimum value of objective function:     0.50000
Number of iterations performed: 7

Done.
```

FILES USAGE

The linear program solver has associated with it the following files:

lp.c - this is the source code file for the linear program solver

lp.back - the is a backup copy of lp.c

lp - this is a Sun 3 compatible executable file for the linear program solver

ltest1 - a file containing input data for the trial run

PROGRAM LISTING

To produce an object file from this source code, enter:

On a Sun 3:

```
cc lp.c -f68881 -lm
```

On a Sun 4:

```
cc lp.c -lm
```

```
#include <stdio.h>
#include <math.h>

#define Mmax 20
#define Nmax 20
#define Zero 0.0000001

/*

   Linear Programming Problem Solver
   ---------------------------------

     By: Jim Whitehead
   Date: April 27, 1989


   This program is to be implemented as a procedure for inclusion in
   a larger program. (Will be written as a complete program for testing
   purposes.)

   This program will solve a linear programming problem by use of the
   simplex method. The program solves a linear programming problem expressed
   in standard form:

        minimize     Cx

        subject to   Ax <= b
                      x >= 0

   In this representation, C is a cost vector, A is a constraint matrix,
   b are the constraining values, and x is the variable to be optimized.

   This routine accepts the following input:

   A  - a pointer to an M x N constraint matrix of reals
   b  - a pointer to an M vector of constraining values (reals)
   C  - a pointer to an N vector of costs (reals)
   M  - an integer giving the number of rows in the A matrix
   N  - an integer giving the number of columns in the A matrix
   Zero - an upper bound for zero (to avoid round-off problems with zero)
   x  - a pointer to a matrix which can be used to hold the optimized
        variable values. (reals)
   mc - a ptr to a real used to hold the value of the cost function at minimum
   err - a ptr to a return code used to indicate if the procedure has failed
         (0 indicates proper functioning)
   iter - a ptr to an int that will return the total number of complete
          iterations performed to reach the optimum value
   inbas - a pointer to an array of integers indicating which columns of
           A are to start in the basis. Column numbering starts with 0!
   outbas - a pointer to an array of integers indicating which columns of
            A are to start out of the basis. Column number starts with 0!
            The inbas matrix is N elements long, the outbas matrix is
            M - N elements long.

   This routine uses internally:

   Binv - an M x M matrix containing the inverse of the basis matrix.
   iter - an integer containing a count of the current simplex iteration
   jstar - an integer denoting which column of the non-basis columns of A
           is to join the basis columns of A
```

```
   kstar - an integer denoting which column of the basis will leave the
           basis during this iteration
   tstar - a real giving the amount of travel in the optimizing direction
           to take in this iteration
   d     - an N vector containing the direction of optimality
   i,j,k - loop counters
   sum   - a double used to hold intermediate values during matrix mults.

/*
  Now have functions and declarations for Simplex algorithm:
*/


/*
   lufact - performs LU factorzation on an input array. This
            routine is called from the matrix inverse routine.

   input:
   array - an array of reals of size blksize x blksize
   piv   - pivot array (initially [0,1,2,...N-1])
   blksize - size of array

   output:
   array - contains compact storage form of LU factorization
   piv   - pivot array, modified

*/

int LUFact(array, piv, blksize)

float array[][Nmax];
int piv[];
int blksize;

{
   int i, j, k, p, tp, tpiv;
   double max, temp;

   /* following algorithm on p. 345 of Burden & Faires: */

   /* Step 1 */
   max = array[0][0];
   tp = 0;
   for (p = 1; p < blksize ; p++) {
       if (fabs(array[p][0]) > max) {
           tp = p;
           max = fabs(array[p][0]);
       }
   }
   if (max == 0.0) {
      printf("LUFact: Pivot error (max is zero)");
      exit(1);
   }
   /* Step 2 */
   if (tp != 0) {
       *piv = tp;
       *(piv+tp) = 0;
   }
   /* Step 4 */
   for (j = 1; j < blksize; j++) {
```

```
        array[*(piv+j)][0] = (array[*(piv+j)][0] / array[*piv][0]);
    }
    /* Step 5 */
    for (i=1; i < (blksize-1); i++) {
        /* Step 6 */
        max = 0.0;
        tp = i;
        for (j=i; j < blksize; j++) {
            temp = array[*(piv+j)][i];
            k = 0;
            do {
               temp = temp - array[*(piv+j)][k] * array[*(piv+k)][i];
               }
            while (k++ < (i-1));
            temp = fabs(temp);
            if (temp > max) {
                max = temp;
                tp = j;
            }
        }
        /* Step 7 */
        if (tp != i) {
            tpiv = *(piv+tp);
            *(piv+tp) = *(piv+i);
            *(piv+i)  = tpiv;
        }
        /* Step 8 */
        temp = 0.0;
k = 0;
        do {
            temp += array[*(piv+i)][k] * array[*(piv+k)][i]; /* + OK*/
           }
        while ( k++ < (i-1));
        array[*(piv+i)][i] -= temp;

        /* Step 9 */
        for (j = i+1; j < blksize ; j++) {
            temp = array[*(piv+i)][j];
            k = 0;
            do {
               temp -= array[*(piv+i)][k] * array[*(piv+k)][j];
               }
            while (k++ < (i-1));
            array[*(piv+i)][j] = temp;

            temp = array[*(piv+j)][i];
            k = 0;
            do {
               temp -=  array[*(piv+j)][k] * array[*(piv+k)][i];
               }
            while ( k++ < (i-1));
            array[*(piv+j)][i] = temp / array[*(piv+i)][i];
         }
    }
    /* Step 10 */
    temp = array[*(piv+blksize-1)][blksize-1];
    for (k=0; k < (blksize-1); k++)
        temp -= array[*(piv+blksize-1)][k] * array[*(piv+k)][blksize-1];
    if (temp == 0.0 ) {
        printf("LUFact: Step 10 -- No unique solution exists.");
```

```
        exit(1);
    }
    array[*(piv+blksize-1)][blksize-1] = temp;
} /* End lufact */

/*
 LUSolv - a function to solve the system of equations: LUx = b,
          where L and U are lower and upper triangular matricies
          and x and b are column vectors.

 Input:
        Mat - a block matrix containing L and U stored in compact form
              (i.e., not storing diagonals of L matrix)
        piv - a pivot value matrix (from LUFact routine)
        blksize - the size of Mat
        b - the b in LUx = b
        x - the x in LUx = b (function output)
*/

void LUSolv(Mat, piv, blksize, b, x)

float Mat[][Nmax];   /* Holds LU factored matrix */
int piv[];           /* pivot values matrix      */
int blksize;         /* tells how big Mat is     */
float b[];           /* from LUx = b             */
float x[];           /* from LUx = b (output)    */

{
  float z[Mmax];          /* Used for intermediate value storage */
  int i,j;               /* loop counters */
  float dec;              /* temp. storage variable */

  z[0] = b[0];

  for (i=1; i < blksize; i++) {
     dec = *(b+i);
     for (j=0; j <= (i-1); j++)
        dec -= Mat[*(piv+i)][j] * *(z+j);
     if ( fabs(dec) < Zero )
        dec = 0.0;
     *(z+i) = dec;
     }

  x[blksize-1] = z[blksize-1] / Mat[*(piv+blksize-1)][blksize-1];

  for (i=blksize-2; i>= 0 ; i--) {
     dec = *(z+i);
     for (j=i+1; j<blksize; j++)
        dec -= Mat[*(piv+i)][j] * *(x+j);
     if ( *((*(Mat+i))+i) )
        dec = dec / Mat[*(piv+i)][i];
     if ( fabs(dec) < Zero )
        dec = 0.0;
     *(x+i) = dec;
     }
}



/*
```

```
 Invert - a function to calculate the inverse of a block matrix

 Input:
        Mat - matrix to be inverted
        Matinv - the inverted matrix
        blksize - the size of each matrix

*/

void Invert(Mat, Matinv, blksize)

float Mat[][Nmax];      /* the matrix to be inverted */
float Matinv[][Mmax];   /* the inverted matrix       */
int blksize;            /* the size of the matricies */

{
  int piv[Mmax];  /* a pivot matrix used in the LUfact routine */
  float tc[Mmax];  /* a temporary vector used in calc. of inverse */
  float tr[Mmax];  /* a temporary results vector used in calcs. */
  int i, j;

  int LUFact();
  void LUSolv();

  for (i=0; i < blksize; i++)                    /* init piv matrix */
     *(piv+i) = i;

  LUFact(Mat, piv, blksize);                     /* perform LU factorization */

/*
 To find inverse, repeatedly solve LU system for columns from the
 identity matrix, and place results into the inverse matrix.
*/

  for (i=0; i < blksize; i++) {
     for (j=0; j < blksize; j++)                 /* Set-up identity matrix */
        *(tc+j) = 0.0;
     *(tc+i) = 1.0;
     LUSolv(Mat, piv, blksize, tc, tr);          /* Now solve */
     for (j=0; j < blksize; j++)                 /* Copy results into Matinv */
        Matinv[i][j] = *(tr+j);
     }
} /* Invert */

/*

  PrimSimp - a routine to perform the Simplex algorithm of solving
             linear programming problems. A sophisticated LP solver
             would call this routine once for a Phase 1 solution,
             and then again for a Phase 2 solution. This routine
             solves the primal Simplex problem.

  Input:
        A - constraint matrix
        b - constraint values vector
        C - costs vector
        x - objective variables (an output)
        Binv - an inverted basis matrix if known to the calling routine,
               an empty space if unknown to the calling routine.
```

```
        *mc - will hold the value of the objective function (an output)
        inbas - integers specifying which columns of A are initially in
                the basis
        outbas - same as inbas, but for vectors out of the basis
        M, N - the size of the constraint matrix
        *err - will return an error code, if needed
        *iter - the number of complete iterations performed to find
                optimum value
        Bflag - Set to 1 if Binv already known, set to 0 if Binv should
                be calculated initially.
        Xflag - Set to 1 if x vector already known, set to 0 if x should
                be calculated initially.


*/

void PrimSimp(A,b,C,x,Binv,mc,inbas,outbas,M,N,err,iter, Bflag, Xflag)

float A[][Nmax];              /* constraint matrix       */
float b[];                    /* constraining values     */
float C[];                    /* costs vector            */
float x[];                    /* variables matrix        */
float Binv[][Mmax];           /* matrix inverse of basis */
float *mc;                    /* minimum cost (optimum)  */
int inbas[];                  /* columns in basis        */
int outbas[];                 /* columns not in basis    */
int M,N;                      /* A is an M x N matrix    */
int *err;                     /* used to return errors   */
int *iter;                    /* number of iterations    */
int Bflag;                    /* do/don't calc. Binv     */
int Xflag;                    /* do/don't calc. x vector */

{

  int jstar;                 /* which column is to enter basis */
  int kstar;                 /* which column will leave basis  */
  float tstar;               /* how far in optimum direction to travel */
  float d[Nmax];             /* optimum direction vector       */
  int i,j,k;                 /* loop counters                  */
  float sum, min, mostneg, t, t1, t2;   /* temporary variables */
  float selbas[Nmax];        /* used to hold a calculated vector */
  float temp[Mmax][Mmax];    /* a temp matrix to calc. Binv    */
  float temp2[Mmax];         /* a temp vector                  */
  float rat;                 /* a temp ratio variable          */

  void Invert();

/*
  First, calculate Binv from scratch.
  Copy the columns of A that belong in the matrix into a temporary
  matrix that is then used to call the Inverse function.
*/

  *iter = 0;                              /* Initialize variables */
  *err = 0;
  *mc = 0.0;


  if (!Bflag) {
     for (i=0; i < M ; i++)                /* Copy basis into temp */
```

```
        for (j=0; j < M ; j++)
           temp[i][j] = A[i][*(inbas+j)];

     Invert(temp,Binv,M);                  /* Now have Binv */
     }

/*
  Second, calculate the current x matrix from:

                  x = [Binv*b,0]
*/

  if (!Xflag) {
     for (i=0; i < N; i++)                 /* Initialize all of x to 0.0 */
        *(x+i) = 0.0;

                                           /* Now perform matrix mult */
     for (i=0; i < M; i++) {
        sum = 0.0;
        for (j=0; j < M; j++) {
           t1 = Binv[i][j];                /* Binv[i][j] */
           t2 = *(b + j);                  /* b[j] */
           if (t1 && t2)
             sum += t1*t2;
           } /* for j */
        *(x + *(inbas+i)) = sum;           /* x[inbas[i]] = sum */
     } /* for i */
  } /* if */

/*
  Third, calculate Cn - Cb * Binv * N and place result into selbas vector.
  This vector will then be used to determine which non-basis variable is
  to enter the basis. (In the next step)
*/

Loop:                                      /* A label -- if the LP isn't yet
                                              optimal, program flow returns
                                              here for next loop           */

(*iter)++;

/*
  a) Calculate Cb * Binv = temp2
*/

  for (i=0; i<M; i++) {
     sum = 0.0;
     for (j=0; j<M; j++) {
        t1 = *(C + *(inbas+j));            /* C[inbas[j]] */
        t2 = Binv[j][i];                   /* Binv[j][i]  */
        if (t1 && t2)
          sum += t1*t2;
        }
     *(temp2 + i) = sum;                   /* temp2[i] */
     }

/*
  b) Calculcate Cn - temp2 * N = selbas
*/
```

```
 for (i=0; i < (N-M); i++) {
    sum = 0.0;
    for (j=0; j < M; j++) {
       t1 = *(temp2+j);                    /* temp2[j] */
       t2 = A[j][*(outbas+i)];             /* A[j][outbas[i]] */
       if (t1 && t2)
         sum += t1*t2;
       }
    *(selbas+i) = *(C + *(outbas+i)) - sum;  /* selbas[i] = C[outbas[i]]-sum *
    }

/*
  Fourth, look at the selbas matrix. Are all of the elements greater than zer
  If so, the problem has been optimized, and program flow jumps to the end
  of the function, at the Optimized label. Otherwise, find the most negative
  element, and store location of this element in jstar. The variable
  corresponding to jstar will enter the basis (in a following step).
*/

  mostneg = 0.0;
  jstar = 0;

  for (i=0; i < (N-M); i++)
     if ( *(selbas+i) < mostneg) {
        mostneg = *(selbas+i);
        jstar = i;
        }
  if (!mostneg) goto Optimized;

/*
  Fifth, since the LP is not yet optimized, calculate d, the direction
  of optimization for the LP. This direction can be found by:

          d = [-Binv*N.jstar, 0]
*/

  for (i=0; i < M; i++) {
     sum = 0.0;
     for (j=0; j < M ; j++) {
        t1 = - Binv[i][j];                 /* -Binv[i][j]   */
        t2 = A[j][*(outbas+jstar)];  /* A[j][outbas[jstar]] */
        if (t1 && t2)
          sum += t1*t2;
        }
     if (fabs(sum) > Zero)
       *(d + *(inbas + i)) = sum;             /* d[inbas[i]] */
     else
       *(d + *(inbas + i)) = 0.0;             /* d[inbas[i]] */
     }

                                          /* Set non-basis equal to */
                                          /* identity matrix column */
                                          /* jstar.                 */
  for (i=0; i < (N-M) ; i++)
     *(d + *(outbas+i)) = (i != jstar) ? 0.0 : 1.0;    /* d[outbas[i]] */

/*
  Sixth, take a look at the d matrix elements belonging to the basis.
  Are all of these values greater than or equal to zero? If so, the LP is
  unbounded, so set *err equal to 2, and go to the label Error.
```

```
*/

  for (i=0; i < M; i++)
     if (*(d + *(inbas+i)) < 0.0) goto Findmin;

  *err = 2;
  goto Error;

/*
  Seventh step. Since the LP is not unbounded, it is desirable to
  calculate how far the program should progress in the optimal direction.
  This is accomplished by looking at the following ratio for all
  variables in the basis:

              tstar = -x/d  (for basis values only)

  The variable which corresponds to the minimum value of this ratio is
  denoted kstar, and represents the variable to exit the basis. The
  value of tstar represents the minimum value of the ratio, and is
  the value of how much the LP progresses in the direction d during
  the present iteration.
*/

Findmin:

  min = 0.0;
  for (i=0; i < M; i++) {
     if (*(d + *(inbas+i)) < 0.0) {
        t1 = - *(x + *(inbas+i));                    /* x[inbas[i]] */
        t2 = *(d + *(inbas+i));                      /* d[inbas[i]] */
        if (t2) t = t1/t2;
        if (!min)
           min = t+1;
        if (t < min) {
           min = t;
           kstar = i;
         }
        }
     }
   tstar = min;

/*
  Eighth step. Now that we know which variables are to enter and leave
  the basis, perform the pivot operation. Pivoting includes swapping
  the variables, swapping the columns in A associated with the
  swapped variables, and updating the Binv matrix.
*/

/*
  a) Update Binv matrix.
*/

  for (i=0; i < M; i++) {
     if (i != kstar) {
        rat = ( *(d + *(inbas+i)) / *(d + *(inbas+kstar)) );
        for (j=0; j < M; j++) {
           t = Binv[i][j] - rat * Binv[kstar][j];
           if (fabs(t) < Zero)
             temp[i][j] = 0.0;
           else
```

```
            temp[i][j] = t;
          }
        }
      else {
        rat = 1 / *(d + *(inbas+kstar));
        for (j=0; j < M; j++) {
          t = -rat * Binv[kstar][j];
          if (fabs(t) < Zero)
            temp[i][j] = 0.0;
          else
            temp[i][j] = t;
          }
        }
      }

                                                  /* Now copy temp back */
      for (i=0; i < M; i++)
        for (j=0; j < M; j++)
          Binv[i][j] = temp[i][j];

/*
  b) Swap variables and columns by swapping values in inbas and outbas.
     The idea is to replace column kstar of B with column jstar of N.
*/

  i = inbas[kstar];                 /* Save value for swap */
  inbas[kstar] = outbas[jstar];     /* Bring kstar into basis */
  outbas[jstar] = i;                /* Bring jstar into N */


/*
  Ninth (and final) step. Since the direction of increasing optimality
  has been calculated, and since the amount of travel in that direction
  has been calculated, recalculate x to reflect these facts. Thus,

                    x̄' = x̄ + tstar * d̄

*/

  for (i=0; i < N; i++) {
    *(x+i) = *(x+i) + tstar * *(d+i);
    if (*(x+i) < Zero)
      *(x+i) = 0.0;
    }

/*
  Now loop.
*/

  goto Loop;

/*
  Now handle case where the LP has been optimized, and calculate the
  final value of the objective function. Subtract iter to reflect number
  of complete iterations performed to find optimum.
*/

Optimized:

  *mc = 0.0;
```

```
    for (i=0; i<N; i++)
       *mc += *(C+i) * *(x+i);

  /*
    Program flow now flows through this error point, and back to the user.
    If no errors have been made, flowing through this error point creates
    no problems.
  */

  Error:

   i++;    /* A do-nothing to make the compiler happy */

  } /* PrimSimp */


  /*

    ConvertToStandard

    Accepts as input the following:

    o  A,  a matrix of constraint equations for a linear program
    o  eq, stating that the first eq rows of A represent equality
           constraints
    o  lt, stating that the next lt rows of A represent less than
           or equal to inequality constraints
    o  gt, stating that the remaining gt rows of A represent
           greater than or equal to inequality constraints
    o M, N two variables denoting the size of A. A is M x N. The
           variable N will be changed as columns are added.

    This routine then performs the following operations:

    o  For equality constraints, leave the equations as-is. They
       are already in standard form.

    o  For less than or equal to type constraints, add a column
       representing a slack variable to A. This new column will
       be an appropriate column from the identity matrix.

    o  For greater than or equal to constraints, add a column
       representing a salck variable to A. This new column will
       be the negative of the appropriate column from the identity
       matrix.
  */

  void ConvertToStandard(A,eq,lt,gt,M,N)

  float A[][Nmax];                /* Constraint matrix              */
  int eq;                         /* # of equals constraints        */
  int lt;                         /* # of <= constraints            */
  int gt;                         /* # of >= constraints            */
  int *M,*N;                      /* Size of A                      */

     int i,j;                    /* loop variables                 */

     for (i=eq; i < (eq+lt); i++) {       /* less than, so add slacks */
```

```
        for (j=0; j < *M; j++)
           A[j][*N] = (j == i) ? 1.0 : 0.0;
        (*N)++;
        }

    for (i=(eq+lt); i < (eq+lt+gt); i++) {  /* greater than, so add */
        for (j=0; j < *M; j++)              /* negative slacks      */
            A[j][*N] = (j == i) ? -1.0 : 0.0;
        (*N)++;
        }
} /* ConvertToStandard */



/*

  Simplex

  A routine to perform the Simplex algorithm on a given minimization
  problem. (Linear Programming problem). The program accepts the
  problem in the following form:

              Minimize/Maximize Cx

              Subject to:       Ax (=, <=, >=) b

                                 x >= 0   (always)

              where  o  the first eq rows of A are equations in the
                        form Ax = b
                     o  the next lt rows of A are equations in the
                        form Ax <= b
                     o  the next gt rows of A are equations in the
                        form Ax >= b

              and where    o  A is M x N
                           o  b is 1 x M
                           o  C is 1 x N

  The routine then converts the problem into standard form,
  and performs a two phase solution to the problem. Phase 1 finds an
  initial basis for the problem, and Phase 2 solves the problem from
  this initial basis.

*/

void Simplex(A,b,C,x,M,N,Mflag,eq,lt,gt,mc,iter,err)

float A[][Nmax];                /* Constraint matrix                */
float b[];                      /* Constraint values matrix         */
float C[];                      /* Costs matrix                     */
float x[];                      /* Objective variable values        */
int   M, N;                     /* A is an M x N matrix             */
int   Mflag;                    /* If 0, minimize. If 1, maximize.  */
int   eq;                       /* First eq of A are Ax = b         */
int   lt;                       /* Next lt of A are Ax <= b         */
int   gt;                       /* Last gt of A are Ax >= b         */
float *mc;                      /* Used to return the min/max value */
int   *iter;                    /* Returns total number of iterations */
int   *err;                     /* Returns an error code if necessary */
```

```
{
  float Binv[Mmax][Mmax];     /* Inverse of basis matrix                */
  int   inbas[Nmax];          /* Used to specify columns in the basis   */
  int   outbas[Mmax];         /* Used to specify columns not in basis   */
  int   Bflag;                /* Used to specify whether to calc. Binv. */
  int   Xflag;                /* Used to specify whether to calc. x     */
  float e[Nmax];              /* Used in phase1 calculations            */
  int   Mnew, Nnew;           /* Used in each phase's calculations      */
  int   Titer;                /* A temporary iter value holder          */

  float t1,t2;                /* Temporary storage                      */
  float sum;                  /* A temp. storage used for mat. mults.   */
  int i,j,k;                  /* loop counters                          */
  int temp,idflag;            /* temporary storage                      */
  int artif;                  /* a variable to keep track of where the  */
                              /* artificial variables begin in A        */

  void PrimSimp();            /* Primal Simplex Routine                 */
  void ConvertToStandard();   /* Convert input A into standard Ax = b   */

/*
  Initialize variables:
*/

  *mc    = 0;
  *iter  = 0;
  *err   = 0;
  Titer  = 0;
  Mnew   = M;
  Nnew   = N;

  for (i=N; i < Nmax; i++) {  /* Make sure all non-used C values are 0 */
     C[i] = 0;
     inbas[i] = 0;             /* Initialize in-basis vector      .     */
     e[i] = 0;                 /* Initialize auxiliary costs vector     */
     x[i] = 0;                 /* Initialize x vector                   */
     }

  for (i=0; i < Mmax; i++)    /* Initialize out of basis vector         */
     outbas[i] = 0;

  /* Now convert input problem into standard form. */

     if (Mflag)                  /* If a max problem, convert to a min */
        for (i=0; i < N; i++)
           C[i] = - C[i];

     /* Connvert all less than or greater than constraints to = type */

     ConvertToStandard(A,eq,lt,gt,&M,&Nnew);

     /* Check for values of b matrix that are less than zero */

     for (i=0; i < M; i++)       /* Equals constraint */
        if (b[i] < 0) {             /* if b[i] < 0, mult by -1. Remains */
           for (j=0; j < Nnew; j++) /* an equality constraint           */
              A[i][j] = -A[i][j];
           b[i] = - b[i];
           if ((i >= eq) && (i < (eq+lt))) {
```

```
            lt--;
            gt++;
            }
         if ((i >= (eq+lt)) && (i < (eq+lt+gt))) {
            gt--;
            lt++;
            }
         } /* if & hence for */

/* Now create phase 1 problem. For all less than or equals type
   constraints, one column in the basis is already known. For
   all unknown basis columns, create an appropriate column from
   the identity matrix. It is desired to have an identity matrix
   as an initial basis for the phase 1 problem. A new costs vector
   is used, comprised of zeros for all columns from less than
   constraints, and ones from all added (artificial) variable
   columns.                                                    */

   for (i=(Nnew-gt-lt); i < Nnew; i++) {
      temp = -1;
      idflag = 0;
      for (j=0; j < M;  j++) {
         if (A[j][i] == 1.0)
            if (inbas[j] == 0) {
               if (!idflag) {
                  temp = j;
                  idflag = 1;
                  } /* if */
            } /* if */
         } /* for j */
      if (temp != -1)
         inbas[temp] = i;
      }

   /* Now have collected all lt terms. Add artificials to make up. */

   artif = Nnew;            /* keep track of where arificials start */

   for (i=0; i < M; i++)
      if (inbas[i] == 0) {
         for (j=0; j < M; j++)
             A[j][Nnew] = (j == i) ? 1.0 : 0.0;
         inbas[i] = Nnew;
         e[Nnew++] = 1.0;
      }

   /* Now create outbas matrix */

   for (i=0; i < (Nnew-gt-lt); i++)
      outbas[i] = i;

   for (j=i; j < Nnew; j++) {
      temp = 0;
         for (k=0; k < M; k++) {
             if (inbas[k] == j)
                temp = 1;
         } /* for k */
      if (!temp)
         outbas[i++] = j;
      }
```

```
        /* Since the basis matrix is the identity matrix, the inverse
           of the basis matrix is also the identity matrix.          */

        for (i=0; i < M; i++)
           for (j=0; j < M; j++)
              Binv[i][j] = (i == j) ? 1.0 : 0.0;

        Bflag = 1;

        /* Since the basis matrix is the identity matrix, the initial
           x vector is (0,b).                                        */

        for (i=0; i < M; i++)
           x[inbas[i]] = b[i];

        Xflag = 1;

        /* Now we're ready to call PrimSimp to execute phase 1.      */

    PrimSimp(A,b,e,x,Binv,mc,inbas,outbas,M,Nnew,err,&Titer,Bflag,Xflag);

    *iter = Titer;

    temp = 0;
    for (i=artif; i < Nnew; i++)
       if (x[i] != 0.0) temp = 1;

    if (temp) {            /* If artificial variables aren't 0.0, */
       *err = 4;            /* Linear Program is infeasible.       */
       goto SimpErr;
       }


/* Now create Phase 2 problem, based on Phase 1 results. To do this,
   first check the basis from the end of phase 1. If there are no
   artificial variables left in the basis, then remove the artificial
   variables, and solve (the current basis is a feasible basis).
   Otherwise, if some of the artificial variables remain in the
   basis, create the Phase 2 problem as follows. First, change the
   cost function back to C. Next, add a new constraint of the
   form y + z = 0 (y - artificial variable, z - new variable).
   A new basis inverse, and basis feasible solution are then
   calculated. Given this information, the PrimSimp routine is
   again called, this time returning the answer to the original
   problem.                                                        */


/* Check to see if artificial variables are nonbasic. */

temp = 0;
for (i=artif; i < Nnew; i++)
   for (j=0; j < M; j++)
      if (inbas[j] == i)
         temp = 1;

if (!temp) {               /* yes, nonbasic, so remove artificial vars */
   for (i=0; i < (Nnew-Mnew); i++)
      if (outbas[i] >= artif)
        outbas[i] = -1;
```

```
      k = 0;
      for (i=0; i < (Nnew-Mnew); i++)
         if (outbas[i] != -1)
            outbas[k++] = outbas[i];
      Nnew=artif;
      goto DoPhase2;
      }

   /* OK, some artificial variables are basic. Alter A to add new
      constraint and new variable.                                  */

   for (i=0; i < M; i++)        /* Add new column */
      A[i][Nnew] = 0.0;
   A[M][Nnew] = 1.0;

   for (i=0; i < artif; i++)    /* Add new row */
      A[M][i] = 0.0;
   for (i=artif; i <= Nnew; i++)
      A[M][i] = 1.0;
   b[M] = 0.0;

   /* Add new column to the basis */

   inbas[M] = Nnew;

   /* Now update the Binv matrix */

   for (i=0; i < M; i++) {
      sum = 0.0;
      for (j=0; j < M; j++) {
          t1 = -A[M][inbas[j]];
          t2 = Binv[j][i];
          if (t1 && t2)
             sum += t1 * t2;
           } /* for j */
      Binv[M][i] = sum;
      }

   Binv[M][M] = 1.0;

   Nnew++;
   Mnew++;    /* because Mnew = M */

   /* Everything is prepared for phase 2. Start up the Simplex routine */

   DoPhase2:

   PrimSimp(A,b,C,x,Binv,mc,inbas,outbas,Mnew,Nnew,err,iter,Bflag,Xflag);

   *iter += Titer;

   SimpErr:

   i++;    /* bogus instruction to make compiler happy */

} /* Simplex */


/* Main program starts here */
```

```
main()

{


/*
  What follows is a sample variable declaration block for using the
  simplex algorithm.
*/

  float A[Mmax][Nmax];        /* Constraint matrix                          */
  float b[Mmax];              /* Constraint values matrix                   */
  float C[Nmax];              /* Costs matrix                               */
  float x[Nmax];              /* Objective variable values                  */
  float mc;                   /* Used to hold the minimum cost at opt.      */
  int  M, N;                  /* Used to specify size of A actually used    */
  int  eq;                    /* The first gt rows of A are Ax = b          */
  int  lt;                    /* The next lt rows of A are Ax <= b          */
  int  gt;                    /* The last gt rows of A are Ax >= b          */
  int  err;                   /* Returns an error code from Simplex rtn.    */
  int  iter;                  /* Used to find number of iterations          */
  int  Mflag;                 /* Set to 0 if min, 1 if max problem          */

  void Simplex();

  char VarName[Mmax][10];     /* Used to name variables in x vector         */
  int i,j;                    /* Loop counters                              */

/* Start reading input from the user */

  printf("\n\nLinear Program Solver\n");
  printf("   (Simplex Method)\n\n\n");

  M = Mmax;
  N = Nmax;

  while (N >= (Nmax/2)) {
     printf("Please enter the number of variables: ");
     scanf("%d",&N);
     if (N > (Nmax/2))
        printf("\nPlease enter a number in the range (1 - %2d)\n",Nmax/2);
     }

  while (M >= (Mmax/2)) {
     printf("\nPlease enter the number of constraint equations: ");
     scanf("%d",&M);
     if (M > (Mmax/2))
        printf("\nPlease enter a number in the range (1 - %2d)\n",Mmax/2);
     }

  eq = 0;
  lt = 0;
  gt = 0;

  printf("\n\nConstraint equations can be one of three types, either\n");
  printf("equality constraints, less than constraints, or greater than\n");
  printf("constraints. Please enter three numbers below to specify how\n");
  printf("many of each type equation are present in your constraint\n");
  printf("equation matrix.\n");
```

```
while ((gt+lt+eq) != M) {
   printf("\nPlease enter how many equality constraints are present: ");
   scanf("%d",&eq);

   printf("\nPlease enter how many less than or equal to type\n");
   printf("constraints are present: ");
   scanf("%d",&lt);

   printf("\nPlease enter how many greater than or equals to type\n");
   printf("constraints are present: ");
   scanf("%d",&gt);

   if ((eq+lt+gt) != M) {
      printf("\n\nAll three values must add up to the number of");
      printf("equations (%2d). Re-enter.\n\n",M);
      }
   }

printf("\n\n\nPlease enter a ten character (or less) description of ");
printf("each variable.\n");

for (i=0; i < N; i++) {
   printf("\nVariable %2d: ",(i+1));
   scanf("%10s",VarName[i]);
   }

printf("\n\n\nFor each constraint equation, please enter all ");
printf("variable coefficients below:\n");

if (eq)
   printf("\n\nEquality Constraints:\n");
for (i=0; i < eq; i++) {
   printf("\nEquation #%2d: \n\n",(i+1));
   for (j=0; j < N; j++) {
      printf("Coefficient of %s : ",VarName[j]);
      scanf("%f",&A[i][j]);
      }
   printf("\nConstraining value: ");
   scanf("%f",&b[i]);
   }

if (lt)
   printf("\n\nLess than or equal to Constraints:\n");
for (i=eq; i < (eq+lt); i++) {
   printf("\nEquation #%2d: \n\n",(i+1));
   for (j=0; j < N; j++) {
      printf("Coefficient of %s : ",VarName[j]);
      scanf("%f",&A[i][j]);
      }
   printf("\nConstraining value: ");
   scanf("%f",&b[i]);
   }

if (gt)
   printf("\n\nGreater than or equal to Constraints:\n");
for (i=(eq+lt); i < (eq+lt+gt); i++) {
   printf("\nEquation #%2d: \n\n",(i+1));
   for (j=0; j < N; j++) {
      printf("Coefficient of %s : ",VarName[j]);
```

```
         scanf("%f",&A[i][j]);
         }
      printf("\nConstraining value: ");
      scanf("%f",&b[i]);
      }

   printf("\n\n\nFor each variable, please enter an associated cost ");
   printf("to minimize.\n");

   for (i=0; i < N; i++) {
      printf("\nPlease enter the cost for %s : ",VarName[i]);
      scanf("%f",&C[i]);
      }

   /* Input complete */

   Mflag = 0;  /* Assumed minimization */

   Simplex(A,b,C,x,M,N,Mflag,eq,lt,gt,&mc,&iter,&err);

   switch (err)
         {
         case 2  :  printf("\nSimplex: LP unbounded.\n\n");
                    break;

         case 4  :  printf("\nSimplex: LP infeasible.\n\n");
                    break;

         default :  printf("\nSimplex: LP Optimized.\n\n");
         }

   if (err) exit(err);

   printf("Final variable values:\n\n");

   for (i=0; i < N; i++)
      printf("\n%s : %10.5f\n",VarName[i],x[i]);

   printf("\nMinimum value of objective function: %10.5f\n",mc);

   printf("\nNumber of iterations performed: %d\n",iter);

   printf("\n\nDone.\n");
}
```

Quadratic Programming Algorithm

Gradient Projection Method

The quadratic programming algorithm has been implemented by following a modified Gradient Projection Method prepared by Kostas Kyriakopoulos. (Modified from a presentation in Linear and Nonlinear Programming, Second Edition, by David Luenberger.) In this method, the Gradient Projection Method is optimized for use with a quadratic objective function of the form $f(\mathbf{x}) = \mathbf{x}'\mathbf{Q}\mathbf{x}$. This function solves the following programming problem:

minimize $\mathbf{x}'\mathbf{Q}\mathbf{x}$

subject to: $\mathbf{A}\mathbf{x} = \mathbf{b}$ (must be in this form)

For robotic applications, this method is used to minimize the euclidian norm (distance) between a robot and an object in its surroundings.

The basic philosophy of the gradient projection method is as follows. The method starts with a basic feasible solution $\mathbf{x}$. At this point, some of the elements of $\mathbf{A}\mathbf{x}$ will equal $\mathbf{b}$, while others won't. The gradient projection method takes those rows of $\mathbf{A}$ corresponding to those elements of $\mathbf{A}\mathbf{x}$ which equal $\mathbf{b}$ and calls the collection of the indicies of those rows the Working Set. The rows not in the working set form the non-working (or lazy) set. At each step of the gradient projection method, either movement is made (subject to the constraints of the working set) towards the point of optimality, or a constraint is loosened so as to allow movement towards the optimal point in a later step. Steps are repeated until the Langrangian vector for the system is positive, indicating that the optimal position has been reached.

Each step performs the following operations:

1. Calculate $\mathbf{A}\mathbf{x}$. Collect the indicies of the rows for which $\mathbf{A}\mathbf{x} = \mathbf{b}$, and call this collection the working set $W(\mathbf{x})$. The matrix

formed by all of the rows in the working set will be denoted A_q. Collect the indicies of all rows not in the working set and place them in the non-working set, L(x). The matrix formed by all of the rows in L(x) will be denoted A_L.

2. Calculate the matrix **AqAqInv** which is found from the equation: **AqAqInv** = $(\mathbf{Aq}\ \mathbf{Aq}')^{-1}$. Calculate the projection matrix, **P**, from the equation, **P** = **I** - **Aq'** * **AqAqInv** * **Aq**. Action by **P** on any vector yields the projection of that vector onto the working set subspace.

3. Calculate the direction of increased optimality, **d**, from the equation: **d** = -**P** * **Q** * **x**.

4. If **d** 0, calculate Alpha2, the amount to move in the direction of increased optimality, as follows: First, calculate Alpha1 = min $(b_L - A_L * x) / (A_L * D)$ for Alpha1 > 0. Next, calculate Alpha2Candidate as follows: Alpha2Cand = **x'QPQx/x'QPQPQx**. Compare Alpha2Candidate to Alpha1. Alpha2 is the lesser of the two. Now update **x** by calculating **x** = **x** + Alpha2 * **D**. If Alpha1 = Alpha2, a new constraint will be added to the working set, so GOTO 1. Otherwise, find a new direction of increased optimality from the new point, hence GOTO 3.

5. If **d** = 0, calculate the langrangian vector for the system, as follows: **Lambda** = - **AqAqInv** * **Aq** * **Q** * **x** . If **Lambda** >= 0, stop -- the optimal point has been reached. Otherwise, find the most negative element of **Lambda** and delete the corresponding row from the working set, adding it to the non-working set. GOTO 2.

Two source codes exist for the quadratic program solver. The first one, **gpm.c**, was implemented using "C" arrays, which are somewhat wasteful of memory, but which produce much cleaner looking code. The second one, **fly.c** (from creating variables on-the-fly, or the movie The Fly -- be afraid, be very afraid of all the ugly pointer arithmetic) uses the system function calloc() to use only however much memory is actually needed to hold all of the arrays used in the program. When making changes, it would probably be easier to first modify **gpm.c**, and then transfer changes over into **fly.c**.

A Debug option is available when calling the function Minimize. This option enables code that prints the working set, $\mathbf{A_q}$, the non-working set, $\mathbf{A_L}$, **P**, **D**, and either Alpha1, Alpha2Cand, and Alpha2 or **Lambda**, and the minimum value position in **Lambda**. This code has proven to be very useful in debugging the gradient projectoin method, and is highly recommended for use in analysis of how the gradient projection method actually works.

This code is highly segmented, and well documented -- someone familiar with the gradient projection method should have no trouble in analyzing and changing this program.

This code was used with two sets of trial data to ascertain how fast the gradient projection method works. The first set of data was a formulation of the same problem used as a benchmark for the linear programming algorithm. For the quadratic programming routine, however, minimizing the euclidian norm 200 times took 24 seconds. Since each minimization took 9 iterations, the quadratic algorithm achieved 75 iterations/second, and 8.3 minimizations/sec. A second set of data was also ued to benchmarch the quadratic programming algorithm, with the result that 200 minimizations, yielding 1400 iterations, took 24 seconds. This gave a value of 58 iterations/second, and 8.3 minimizations/second. Given that a real-world use would begin near an almost-optimal point, an estimate on the number of objects this routine could minimize for in one second is 15 objects.

FUNCTION DESCRIPTIONS

The following is a listing and brief description of every function in the fly.c and gpm.c code. If more information about these routines is desired, please consult the well-documented source code.

LUFact

The LUFact routine performs an LU factorization on an input array. The basic concept is that, in cases where gaussian elimination is to be repeatedly performed on a block matrix (such as during the calculation of a matrix inverse), it is easier to separate the steps of gaussian elimination and backwards solution. This spares the computational expense of repeated gaussian eliminations on the same matrix (the end result will be the same each time). An LU factorization performs this gaussian elimination step by finding the two matricies **L** & **U** such that a block matrix **A** (as in **Ax** = **b**) can be represented as **LU**. The LU factorization is stored in the original matrix. This is accomplished by assuming the diagonal elements of **L** are all 1.

The method of LU factorization used includes pivoting. Instead of swapping elements in **A** to accomplish this swapping operation, indicies in a pivot matrix are swapped. Thus, in addition to **A**, this routine needs a pivot matrix (internally called piv) initially set to equal [0, 1, 2, ... blksize]. The LUFact routine will change these values if pivoting is necessary.

The actual method implemented comes from pages 345-346 of Numerical Analysis, Third Edition by Richard Burden and Douglas Faires, and references to step numbers in comments in the code refer back to this source.

Input:

array - an array of size blksize x blksize on which to perform LU factorization
piv - a vector of pivot values, as described above. Input values should be [0, 1, 2, ... blksize].
blksize - the size of the block matrix array

Output:

array - in LU factored form. Assumes diagonals of L are 1.
piv - reflects row pivoting of array performed during LU factorization.

LUSolv

Once an array is in the form **A** = **LU**, the system **Ax** = **b** can be solved by finding **z** such that **Lz** = **b**, and then by finding **x**, such that **Ux** = **z**. This routine performs this backwards solution routine.

Input:

Mat - the output of LUFact on a matrix
piv - the result of pivot ooperations in LUFact
blksize - the size of **A**
b - the **b** in **Ax** = **b**

Output:

x - the **x** in **Ax** = **b**, solved.

Invert

A matrix inverse is found by repeatedly solving the system **Ax** = **b** for **b** equal to columns from the identity matrix, and with **x** returning the appropriate column in the matrix inverse. Since this requires the repetitive solving of systems involving **A**, this matrix is LU decomposed before repeatedly calling LUSolv. This makes this a very speedy implementation of the matrix inverse function.

Input:

Mat - the array to be inverted
blksize - the size of Mat is blksize x blksize

Output:

Matinv - the matrix inverse of Mat

MatMult

A function to perform matrix multiplication. C = A * B

Input:

A - first matrix (m x n)
B - second matrix (n x p)
m,n,p - matrix sizes

Output:

C - C = A * B (m x p)

VectMult

A function to multiply a matrix times a vector. c = A * b.

Input:

A - a matrix (m x n)
b - a vector (n x 1)
m,n - matrix and vector sizes

Output:

c - c = A * b (m x 1)

DotProd

A function that returns as its value the dot product of two vectors, DotProd = **a** . **b**.

Input:

a - a vector (m x 1)
b - another vector (m x 1)
m - the length of the vectors

Output:

DotProd - returns the result of **a** . **b**

FindWorkingSet

A function that determines the current working set. It first calculates **Ax**. It then places all those rows for which **Ax** = **b** into the working set, and all non-working set rows are placed into the non-working set.

Input:

A - the constraints matrix (m x n)
b - a constraining values matrix (m x 1)
x - the current position in constraint space
m,n - matrix dimensions

Output:

W - the working set
L - the non-working set
*q - the size of the working set
*r - the size of the non-working set

AddToSet

A function that adds element val to set S, and places val in the proper order within S. Used now for adding an element to the

non-working set.

Input:

val - the element to add to set S
S - the set to add S to
*size - the current size of S

Output:

S - the set, with val added to it
*size - increased by 1

Deactivate

A function that removes a constraint from the working set and adds it to the non-working set.

Input:

val - the element to delete from the working set
W - the working set
L - the non-working set
*q - the size of the working set
*r - the size of the non-working set

Output:

W - the working set, with val removed
L - the non-working set, with val added in place
*q - decreased by one
*r - increased by one

FindAlpha1

A function that calculates the value of Alpha1 from the following equation:

Alpha1 = (min > 0) **b - Ax / A * D**
(for all constraints in the non-working set)

Input:

A - the constraint matrix (m x n)
b - the constraining values matrix (m x 1)
x - the current position in constraint space (n x 1)
D - the direction of increasing optimality (n x 1)
L - the non-working set
m,n - matrix dimensions
r - the size of the non-working set

Output:

FindAlpha1 - returns the value of Alpha1, as calculated

FindAlpha2Cand

A function that calculates the value of Alpha2Cand from the following equation: Alpha2Cand = **x'QPQx / x'QPQPQx.**

Input:

x - the current position in constraint space (n x 1)
Q - the coefficients of the objective function (n x n)
P - the current projection matrix (n x n)
PQ - the result of **P * Q**, previously calculated
n - matrix dimensions

Output:

Alpha2Cand - returns the value of Alpha2Candidate

MakeIdent

A function to create an n x n identity matrix.

Input:

n - the size of the identity matrix to create (n x n)

Output:

I - the n x n identity matrix created

AqAqInverse

This procedure calculates the value of: inv(**Aq** * **Aq'**), where the inv operation is the matrix inverse. This procedure is called with a flag that can constain up to three values: 0 - perform a straight inverse (currently implemented), 1 - perform an inverse update after an operation to add to the working set, and 2 - perform an inverse update after an operation to delete from the working set.

Input:

A - the constraints matrix (m x n)
W - the working set
m,n - matrix dimensions
q - the size of the working set
Flag - described above

Output:

AqAqInv - the value: inv(**Aq** * **Aq'**)

CalcP

This procedure calculates the Projection matrix, **P**, from the following equation: **P** = **I** - **Aq** * **AqAqInv** * **Aq'**.

Input:

A - the constraint matrix (m x n)

I - the identity matrix (n x n)
AqAqInv - the value of inv(**Aq * Aq'**)
W - the working set
m,n - matrix dimensions
q - the size of the working set

Output:

P - the projection matrix, calculated as described above

CalcD

This procedure calculates the direction of increasing optimality vector, **D**, from the following equation: **D = P * Q * x.**

Input:

P - the projection matrix (n x n)
Q - the coeffients of the objective function (n x n)
x - the current position in the constraint space
n - matrix dimensions

Output:

D - the direction of increasing optimality
PQ - calculated within the routine, stored for later use

CalcLagrange

This procedure calculates the langrangian vector, **Lambda**, from the following equation: **Lambda = - AqAqInv * Aq * Q * x .**

Input:

A - constraints matrix (m x n)
AqAqInv - holds the value inv(**Aq * Aq'**)
Q - the coefficients of the objective function
x - the current position in the constraint space

W - the working set
m,n - matrix dimensions
q - the size of the working set

Output:

Lambda - the langrangian vector, calculated as described above

EvalLambda

This function evaluates the vector of the Langrangians. If the vector is greater than or equal to 0, then the pos flag is set to 1, otherwise it is set to 0. If the pos flag is 0, the minimum negative value of the Langrangian vector will be found, and its position returned in MinPos.

Input:

Lambda - the Lagrangian vector
n - the size of the Lagrangian vector

Output:

*pos - if **Lambda** >= 0, pos=1, otherwise pos=0
*MinPos - the location of the most negative member of **Lambda**

CalcObj

This function returns the value of the objective function, **x'Qx**.

Input:

Q - the coefficient of the objective function (n x n)
x - the current position in constraint space (n x 1)
n - matrix dimensions

Output

CalcObj - returns the value of the objective function, **x'Qx**

Update_x

This procedure updates the value of **x**, as follows: **x** = **x** + Alpha2 * **D**.

Input:

x - the current position in the constraint space (n x 1)
Alpha2 - the amount of direction **D** to add to **x**
D - the direction of increasing optimality (n x 1)
n - matrix dimension

Output:

x - the new position of x in the constraint space

PrintLambda

This procedure prints the Lagrangian vector, and the information EvalLambda found out about the Lagrangian vector. Used in Debug mode.

Input:

Lambda - the Lagrangian vector
n - the length of the Lagrangian vector
pos - from EvalLambda
MinPos - the position of the most negative element of Lambda

Output:

This procedure prints the Lagrangian vector, the information in pos, and MinPos.

PrintStatus

A procedure to print the working set, the equations in the working set, the non-working set, the equations in the non-working set, the current position in constraint space, the projection matrix, and the direction of increased optimality. The current iteration number is also displayed.

Input:

A - the constraint matrix (m x n)
b - the constraining values matrix (m x 1)
Q - the coefficients of the objective function (n x n)
x - the current position in constraint space (n x 1)
W - the working set
L - the non-working set
P - the current projection matrix (n x n)
D - the direction of increasing optimality (n x 1)
*iter - the current iteration number
m,n - matrix dimensions
q - the size of the working set
r - the size of the non-working set

Output:

As above, prints the working set, the equations in the working set, the non-working set, the equations in the non-working set, the current position in constraint space, the projection matrix, and the direction of increasing optimality.

Minimize

A procedure to minimize the function **x'Qx** subject to linear constraints (a quadratic program solver). Uses a modified gradient projection method, as described above. This routine forms the heart of the program -- it is the front end for every other procedure. It should be very easy to follow this code,

since it follows nearly verbatim the earlier discussion about the method used.

Input:

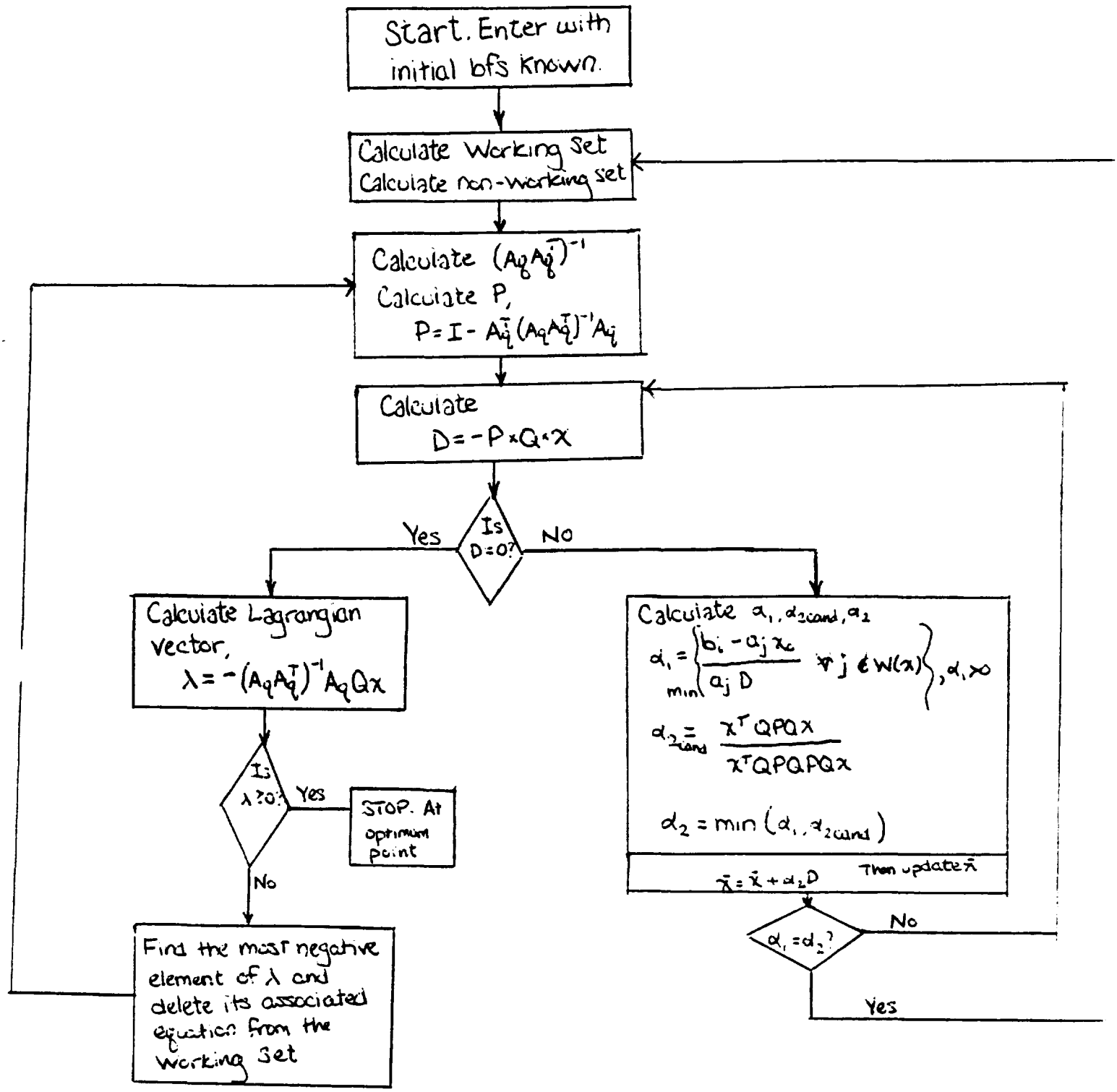
A - the constraint matrix (m x n)
b - the constraining values matrix (m x 1)
Q - the coefficients of the objective function (n x n)
x - the current position in constraint space (n x 1)
m,n - matrix dimensions

Output:

*mc - the minimum value of the objective function
iter - the number of iteratoins it took to find mc
x - the final position vector

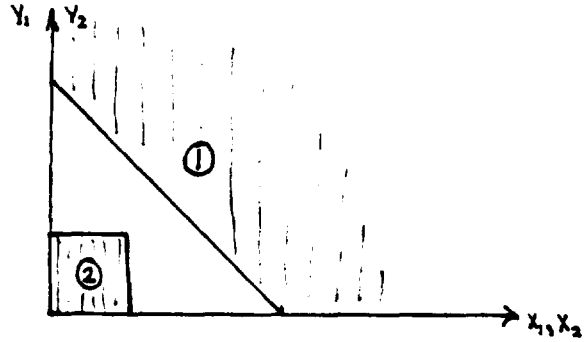
BLOCK DIAGRAM OF THE QUADRATIC PROGRAMMING ALGORITHM

Quadratic Programming / Gradient Projection Method



TRIAL RUN

The following pages contain two trial runs of the quadratic program solver program. The first trial run is executed without Debug mode. The second trial run uses the Debug mode. The physical system being solved is as follows:



This physical system is represented by the following system of equations:

$$\underset{\bar{A}}{\begin{bmatrix} -1 & -1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ -1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & -1 \end{bmatrix}} \underset{\bar{x}}{\begin{bmatrix} x_1 \\ y_1 \\ x_2 \\ y_2 \end{bmatrix}} \leq \underset{\bar{b}}{\begin{bmatrix} -3 \\ 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}}$$

minimize $x'Qx$

$$Q = \begin{bmatrix} 1 & 0 & -1 & 0 \\ 0 & 1 & 0 & -1 \\ -1 & 0 & 1 & 0 \\ 0 & -1 & 0 & 1 \end{bmatrix} \qquad \bar{x}_{initial} = \begin{bmatrix} 3 & 0 & 0 & 0 \end{bmatrix}$$

```
gpm


Quadratic Program Solver
(Gradient Projection Method)


Please enter the number of variables: 4

Please enter the number of constraint equations: 7


Please enter a ten character (or less) description of each variable.

Variable  1: X1

Variable  2: Y1

Variable  3: X2

Variable  4: Y2


For each constraint equation, please enter all variable coefficients below:

Equation # 1:

Coefficient of X1 : -1
Coefficient of Y1 : -1
Coefficient of X2 : 0
Coefficient of Y2 : 0

Constraining value: -3
```

Equation # 2:

Coefficient of X1 : 0
Coefficient of Y1 : 0
Coefficient of X2 : 1
Coefficient of Y2 : 0

Constraining value: 1

Equation # 3:

Coefficient of X1 : 0
Coefficient of Y1 : 0
Coefficient of X2 : 0
Coefficient of Y2 : 1

Constraining value: 1

Equation # 4:

Coefficient of X1 : -1
Coefficient of Y1 : 0
Coefficient of X2 : 0
Coefficient of Y2 : 0

Constraining value: 0

Equation # 5:

Coefficient of X1 : 0
Coefficient of Y1 : -1
Coefficient of X2 : 0
Coefficient of Y2 : 0

Constraining value: 0

Equation # 6:

Coefficient of X1 : 0
Coefficient of Y1 : 0
Coefficient of X2 : -1
Coefficient of Y2 : 0

Constraining value: 0

Equation # 7:

Coefficient of X1 : 0
Coefficient of Y1 : 0
Coefficient of X2 : 0
Coefficient of Y2 : -1

Constraining value: 0

```
Please enter an initial starting point for x from which
minimization will proceed.

Intial value of X1 : 3

Intial value of Y1 : 0

Intial value of X2 : 0

Intial value of Y2 : 0

Please enter a 1 for debugging information, 0 for none: 0


Final variable values:

X1 :    1.50000

Y1 :    1.50000

X2 :    1.00000

Y2 :    1.00000

Minimum value of objective function:    0.50000

Number of iterations performed: 9

Done.
```

```
venus% gpm


Quadratic Program Solver
(Gradient Projection Method)


Please enter the number of variables: 4

Please enter the number of constraint equations: 7



Please enter a ten character (or less) description of each variable.

Variable  1: X1

Variable  2: Y1

Variable  3: X2

Variable  4: Y2


For each constraint equation, please enter all variable coefficients below:

Equation # 1:

Coefficient of X1 : -1
Coefficient of Y1 : -1
Coefficient of X2 : 0
Coefficient of Y2 : 0

Constraining value: -3

Equation # 2:

Coefficient of X1 : 0
Coefficient of Y1 : 0
Coefficient of X2 : 1
Coefficient of Y2 : 0

Constraining value: 1

Equation # 3:

Coefficient of X1 : 0
Coefficient of Y1 : 0
Coefficient of X2 : 0
Coefficient of Y2 : 1

Constraining value: 1
```

```
Equation # 4:

Coefficient of X1 : -1
Coefficient of Y1 : 0
Coefficient of X2 : 0
Coefficient of Y2 : 0

Constraining value: 0

Equation # 5:

Coefficient of X1 : 0
Coefficient of Y1 : -1
Coefficient of X2 : 0
Coefficient of Y2 : 0

Constraining value: 0

Equation # 6:

Coefficient of X1 : 0
Coefficient of Y1 : 0
Coefficient of X2 : -1
Coefficient of Y2 : 0

Constraining value: 0

Equation # 7:

Coefficient of X1 : 0
Coefficient of Y1 : 0
Coefficient of X2 : 0
Coefficient of Y2 : -1

Constraining value: 0


Please enter an initial starting point for x from which
minimization will proceed.


Intial value of X1 : 3

Intial value of Y1 : 0

Intial value of X2 : 0

Intial value of Y2 : 0


Please enter a 1 for debugging information, 0 for none: 1
```

```
==========================================================

        Iteration #  1

Working Set:

[ 0,  4,  5,  6 ]

Working Set of Equations:

[-1.0, -1.0,  0.0,  0.0 : -3.0 ]
[ 0.0, -1.0,  0.0,  0.0 :  0.0 ]
[ 0.0,  0.0, -1.0,  0.0 :  0.0 ]
[ 0.0,  0.0,  0.0, -1.0 :  0.0 ]

Non-Working Set:

[ 1,  2,  3 ]

Non-Working Set of Equations:

[ 0.0,  0.0,  1.0,  0.0 :  1.0 ]
[ 0.0,  0.0,  0.0,  1.0 :  1.0 ]
[-1.0,  0.0,  0.0,  0.0 :  0.0 ]

Current position:

[ 3.000,  0.000,  0.000,  0.000 ]

Current Projection Matrix:

[  0.000,  0.000,  0.000,  0.000 ]
[  0.000,  0.000,  0.000,  0.000 ]
[  0.000,  0.000,  0.000,  0.000 ]
[  0.000,  0.000,  0.000,  0.000 ]

Current Direction Matrix:

[-0.000, -0.000, -0.000, -0.000 ]

Current Value of Objective Function:  9.000


Lagrangian vector:

[ 3.000, -3.000, -3.000, -0.000 ]

Is Lambda positive: No

Most negative element is # 1
```

```
==========================================================

       Iteration #  2

Working Set:

[ 0,  5,  6 ]

Working Set of Equations:

[-1.0, -1.0,  0.0,  0.0 : -3.0 ]
[ 0.0,  0.0, -1.0,  0.0 :  0.0 ]
[ 0.0,  0.0,  0.0, -1.0 :  0.0 ]

Non-Working Set:

[ 1,  2,  3,  4 ]

Non-Working Set of Equations:

[ 0.0,  0.0,  1.0,  0.0 :  1.0 ]
[ 0.0,  0.0,  0.0,  1.0 :  1.0 ]
[-1.0,  0.0,  0.0,  0.0 :  0.0 ]
[ 0.0, -1.0,  0.0,  0.0 :  0.0 ]

Current position:

[ 3.000,  0.000,  0.000,  0.000 ]

Current Projection Matrix:

[  0.500, -0.500,  0.000,  0.000 ]
[ -0.500,  0.500,  0.000,  0.000 ]
[  0.000,  0.000,  0.000,  0.000 ]
[  0.000,  0.000,  0.000,  0.000 ]

Current Direction Matrix:

[-1.500,  1.500, -0.000, -0.000 ]

Current Value of Objective Function:  9.000


Alpha1:  2.000
Alpha2Cand:  1.000
Alpha2:  1.000
```

```
==================================================

       Iteration #  3

Working Set:

[ 0,  5,  6 ]

Working Set of Equations:

[-1.0, -1.0,  0.0,  0.0 : -3.0 ]
[ 0.0,  0.0, -1.0,  0.0 :  0.0 ]
[ 0.0,  0.0,  0.0, -1.0 :  0.0 ]

Non-Working Set:

[ 1,  2,  3,  4 ]

Non-Working Set of Equations:

[ 0.0,  0.0,  1.0,  0.0 :  1.0 ]
[ 0.0,  0.0,  0.0,  1.0 :  1.0 ]
[-1.0,  0.0,  0.0,  0.0 :  0.0 ]
[ 0.0, -1.0,  0.0,  0.0 :  0.0 ]

Current position:

[ 1.500,  1.500,  0.000,  0.000 ]

Current Projection Matrix:

[  0.500, -0.500,  0.000,  0.000 ]
[ -0.500,  0.500,  0.000,  0.000 ]
[  0.000,  0.000,  0.000,  0.000 ]
[  0.000,  0.000,  0.000,  0.000 ]


Current Direction Matrix:

[-0.000, -0.000, -0.000, -0.000 ]

Current Value of Objective Function:  4.500


Lagrangian vector:

[ 1.500, -1.500, -1.500, -0.000 ]

Is Lambda positive: No

Most negative element is # 1
```

==========

Iteration # 4

Working Set:

[0, 6]

Working Set of Equations:

[-1.0, -1.0, 0.0, 0.0 : -3.0]
[0.0, 0.0, 0.0, -1.0 : 0.0]

Non-Working Set:

[1, 2, 3, 4, 5]

Non-Working Set of Equations:

[0.0, 0.0, 1.0, 0.0 : 1.0]
[0.0, 0.0, 0.0, 1.0 : 1.0]
[-1.0, 0.0, 0.0, 0.0 : 0.0]
[0.0, -1.0, 0.0, 0.0 : 0.0]
[0.0, 0.0, -1.0, 0.0 : 0.0]

Current position:

[1.500, 1.500, 0.000, 0.000]

Current Projection Matrix:

[0.500, -0.500, 0.000, 0.000]
[-0.500, 0.500, 0.000, 0.000]
[0.000, 0.000, 1.000, 0.000]
[0.000, 0.000, 0.000, 0.000]

Current Direction Matrix:

[-0.000, -0.000, 1.500, -0.000]

Current Value of Objective Function: 4.500

Alpha1: 0.667
Alpha2Cand: 1.000
Alpha2: 0.667

```
==================================================

       Iteration #  5

Working Set:

[ 0,  1,  6 ]

Working Set of Equations:

[-1.0, -1.0,  0.0,  0.0 : -3.0 ]
[ 0.0,  0.0,  1.0,  0.0 :  1.0 ]
[ 0.0,  0.0,  0.0, -1.0 :  0.0 ]

Non-Working Set:

[ 2,  3,  4,  5 ]

Non-Working Set of Equations:

[ 0.0,  0.0,  0.0,  1.0 :  1.0 ]
[-1.0,  0.0,  0.0,  0.0 :  0.0 ]
[ 0.0, -1.0,  0.0,  0.0 :  0.0 ]
[ 0.0,  0.0, -1.0,  0.0 :  0.0 ]

Current position:

[ 1.500,  1.500,  1.000,  0.000 ]

Current Projection Matrix:

[  0.500, -0.500,  0.000,  0.000 ]
[ -0.500,  0.500,  0.000,  0.000 ]
[  0.000,  0.000,  0.000,  0.000 ]
[  0.000,  0.000,  0.000,  0.000 ]


Current Direction Matrix:

[ 0.500, -0.500, -0.000, -0.000 ]

Current Value of Objective Function:  2.500


Alpha1:  3.000
Alpha2Cand:  1.000
Alpha2:  1.000
```

```
=======================================================

       Iteration #  6

Working Set:

[ 0,  1,  6 ]

Working Set of Equations:

[-1.0, -1.0,  0.0,  0.0 : -3.0 ]
[ 0.0,  0.0,  1.0,  0.0 :  1.0 ]
[ 0.0,  0.0,  0.0, -1.0 :  0.0 ]

Non-Working Set:

[ 2,  3,  4,  5 ]

Non-Working Set of Equations:

[ 0.0,  0.0,  0.0,  1.0 :  1.0 ]
[-1.0,  0.0,  0.0,  0.0 :  0.0 ]
[ 0.0, -1.0,  0.0,  0.0 :  0.0 ]
[ 0.0,  0.0, -1.0,  0.0 :  0.0 ]

Current position:

[ 2.000,  1.000,  1.000,  0.000 ]

Current Projection Matrix:

[  0.500, -0.500,  0.000,  0.000 ]
[ -0.500,  0.500,  0.000,  0.000 ]
[  0.000,  0.000,  0.000,  0.000 ]
[  0.000,  0.000,  0.000,  0.000 ]


Current Direction Matrix:

[-0.000, -0.000, -0.000, -0.000 ]

Current Value of Objective Function:  2.000


Lagrangian vector:

[ 1.000,  1.000, -1.000, -0.000 ]

Is Lambda positive: No

Most negative element is # 2
```

```
==================================================

       Iteration #  7

Working Set:

[ 0,  1 ]

Working Set of Equations:

[-1.0, -1.0,  0.0,  0.0 : -3.0 ]
[ 0.0,  0.0,  1.0,  0.0 :  1.0 ]

Non-Working Set:

[ 2,  3,  4,  5,  6 ]

Non-Working Set of Equations:

[ 0.0,  0.0,  0.0,  1.0 :  1.0 ]
[-1.0,  0.0,  0.0,  0.0 :  0.0 ]
[ 0.0, -1.0,  0.0,  0.0 :  0.0 ]
[ 0.0,  0.0, -1.0,  0.0 :  0.0 ]
[ 0.0,  0.0,  0.0, -1.0 :  0.0 ]

Current position:

[ 2.000,  1.000,  1.000,  0.000 ]

Current Projection Matrix:

[  0.500, -0.500,  0.000,  0.000 ]
[ -0.500,  0.500,  0.000,  0.000 ]
[  0.000,  0.000,  0.000,  0.000 ]
[  0.000,  0.000,  0.000,  1.000 ]


Current Direction Matrix:

[-0.000, -0.000, -0.000,  1.000 ]

Current Value of Objective Function:  2.000


Alpha1:  1.000
Alpha2Cand:  1.000
Alpha2:  1.000
```

==========

Iteration # 9

Working Set:

[0, 1, 2]

Working Set of Equations:

```
[-1.0, -1.0,  0.0,  0.0 : -3.0 ]
[ 0.0,  0.0,  1.0,  0.0 :  1.0 ]
[ 0.0,  0.0,  0.0,  1.0 :  1.0 ]
```

Non-Working Set:

[3, 4, 5, 6]

Non-Working Set of Equations:

```
[-1.0,  0.0,  0.0,  0.0 :  0.0 ]
[ 0.0, -1.0,  0.0,  0.0 :  0.0 ]
[ 0.0,  0.0, -1.0,  0.0 :  0.0 ]
[ 0.0,  0.0,  0.0, -1.0 :  0.0 ]
```

Current position:

[1.500, 1.500, 1.000, 1.000]

Current Projection Matrix:

```
[  0.500, -0.500,  0.000,  0.000 ]
[ -0.500,  0.500,  0.000,  0.000 ]
[  0.000,  0.000,  0.000,  0.000 ]
[  0.000,  0.000,  0.000,  0.000 ]
```

Current Direction Matrix:

[-0.000, -0.000, -0.000, -0.000]

Current Value of Objective Function: 0.500

Lagrangian vector:

[0.500, 0.500, 0.500, -0.000]

Is Lambda positive: Yes

Most negative element is #-1

```
Final variable values:

ζ1 :     1.50000

Y1 :     1.50000

X2 :     1.00000

Y2 :     1.00000

Minimum value of objective function:     0.50000

Number of iterations performed: 9

Done.
```

FILES USAGE

The linear program solver has associated with it the following files:

fly.c - the source code for the quadratic programming algorithm implemented using calloc() based memory saving code

fly.back - a backup copy of fly.c

gpm - the executable file for fly.c (Sun 3 only)

gpm.c - the source code for the quadratic programming algorithm implemented using arrays instead of calloc() based code

gpm.back - a backup copy of gpm.c

gpmII - the executable file for gpm.c (Sun 3 only)

gtest1 - the test file used to produce the sample runs

gtest2 - another test file with a different system of equations

PROGRAM LISTING

To produce an object file from this source code, enter:

On a Sun 3:

```
cc fly.c -f68881 -lm
```

On a Sun 4:

```
cc fly.c -lm
```

```
#include <stdio.h>
#include <math.h>

#define Mmax 10
#define Nmax 10
#define Zero 0.0000001

/*

   Quadratic Programming Problem Solver
   ------------------------------------

     By: Jim Whitehead
   Date: May 13, 1989


   This program is to be implemented as a procedure for inclusion in
   a larger program. (Will be written as a complete program for testing
   purposes.)

   This program will solve a quadratic programming problem by use of the
   gradient projection method. The program solves a quadratic programming
   problem expressed in the following form:

        minimize     x'Cx

        subject to   Ax <= b

   In this representation, C is a quadratic coeffiecients matrix, A is a
   constraint matrix, b are the constraining values, and x is the variable
   to be optimized.

*/

/*

  LUFact - A procedure that accepts as input an array, and provides
           as output two arrays, L & U, such that LU = A (the array).
           A pivot vector is an input (& output). This vectors is used
           to perform pivot operations without any copying of matrix
           elements. Pivoting two rows is performed by swapping indicies
           in the pivot matrix.
*/

int LUFact(array, piv, blksize)

float *array;
int *piv;
int blksize;

{
   int i, j, k, p, tp, tpiv;
   double max, temp;

   /* following algorithm on p. 345 of Burden & Faires: */

   /* Step 1 */
   max = *array;
   tp = 0;
   for (p = 1; p < blksize ; p++) {
```

```
    if (fabs(*(array+p*blksize)) > max) {
       tp = p;
       max = fabs(*(array+p*blksize));
    }
}
if (max == 0.0) {
   printf("LUFact: Pivot error (max is zero)");
   exit(1);
}
/* Step 2 */
if (tp != 0) {
    *piv = tp;
    *(piv+tp) = 0;
}
/* Step 4 */
for (j = 1; j < blksize; j++) {
    *(array+*(piv+j) * blksize) = (*(array+*(piv+j)*blksize) /
                                        *(array+*piv*blksize));
}
/* Step 5 */
for (i=1; i < (blksize-1); i++) {
    /* Step 6 */
    max = 0.0;
    tp = i;
    for (j=i; j < blksize; j++) {
        temp = *(array+*(piv+j)*blksize+i);
        k = 0;
        do {
           temp -= *(array+*(piv+j)+k) * *(array+*(piv+k)*blksize+i);
           }
        while (k++ < (i-1));
        temp = fabs(temp);
        if (temp > max) {
           max = temp;
           tp = j;
        }
    }
    /* Step 7 */
    if (tp != i) {
       tpiv = *(piv+tp);
       *(piv+tp) = *(piv+i);
       *(piv+i)  = tpiv;
    }
    /* Step 8 */
    temp = 0.0;
    k = 0;
    do {
        temp += *(array+*(piv+i)*blksize+k) * *(array+*(piv+k)*blksize+i);
       }
    while ( k++ < (i-1));
    *(array+*(piv+i)*blksize+i) -= temp;

    /* Step 9 */
    for (j = i+1; j < blksize ; j++) {
        temp = *(array+*(piv+i)*blksize+j);
        k = 0;
        do {
         temp -= *(array+*(piv+i)*blksize+k) * *(array+*(piv+k)*blksize+j);
           }
        while (k++ < (i-1));
```

```
            *(array+*(piv+i)*blksize+j) = temp;

            temp = *(array+*(piv+j)*blksize+i);
            k = 0;
            do {
              temp -= *(array+*(piv+j)*blksize+k) * *(array+*(piv+k)*blksize+i);
               }
            while ( k++ < (i-1));
            *(array+*(piv+j)*blksize+i) = temp / *(array+*(piv+i)*blksize+i);
        }
    }
    /* Step 10 */
    temp = *(array+*(piv+blksize-1)*blksize+blksize-1);
    for (k=0; k < (blksize-1); k++)
        temp -= *(array+*(piv+blksize-1)*blksize+k) *
                                          *(array+*(piv+k)*blksize+blksize-1);
    if (temp == 0.0 ) {
       printf("LUFact: Step 10 -- No unique solution exists.");
       exit(1);
    }
    *(array+*(piv+blksize-1)*blksize+blksize-1) = temp;
} /* End lufact */

/*
 LUSolv - a function to solve the system of equations: LUx = b,
          where L and U are lower and upper triangular matricies
          and x and b are column vectors.

 Input:
        Mat - a block matrix containing L and U stored in compact form
              (i.e., not storing diagonals of L matrix)
        piv - a pivot value matrix (from LUFact routine)
        blksize - the size of Mat
        b - the b in LUx = b
        x - the x in LUx = b (function output)
*/

void LUSolv(Mat, piv, blksize, b, x)

float *Mat;          /* Holds LU factored matrix */
int piv[];           /* pivot values matrix      */
int blksize;         /* tells how big Mat is     */
float b[];           /* from LUx = b             */
float x[];           /* from LUx = b (output)    */

{
  float *z;         /* Used for intermediate value storage */
  int i,j;          /* loop counters */
  float dec;        /* temp. storage variable */

  char *calloc();

  z = (float *) calloc(blksize, sizeof(float));

  *z = b[0];

  for (i=1; i < blksize; i++) {
     dec = *(b+i);
     for (j=0; j <= (i-1); j++)
        dec -= *(Mat+*(piv+i)*blksize+j) * *(z+j);
```

```
      if ( fabs(dec) < Zero )
         dec = 0.0;
      *(z+i) = dec;
      }

   x[blksize-1] = *(z+blksize-1) / *(Mat+*(piv+blksize-1)*blksize+blksize-1);

   for (i=blksize-2; i>= 0 ; i--) {
      dec = *(z+i);
      for (j=i+1; j<blksize; j++)
         dec -= *(Mat+*(piv+i)*blksize+j) * *(x+j);
      if ( *(Mat+*(piv+i)*blksize+i) )
         dec = dec / *(Mat+*(piv+i)*blksize+i);
      if ( fabs(dec) < Zero )
         dec = 0.0;
      *(x+i) = dec;
      }

   free(z);

} /* LUSolv */


/*

 Invert - a function to calculate the inverse of a block matrix

 Input:
         Mat - matrix to be inverted
         Matinv - the inverted matrix
         blksize - the size of each matrix

*/

void Invert(Mat, Matinv, blksize)

float *Mat;              /* the matrix to be inverted */
float *Matinv;           /* the inverted matrix       */
int blksize;             /* the size of the matricies */

{
   int *piv;            /* a pivot matrix used in the LUfact routine */
   float *tc;           /* a temporary vector used in calc. of inverse */
   float *tr;           /* a temporary results vector used in calcs. */
   int i, j;

   int LUFact();
   void LUSolv();
   char *calloc();

   piv = (int *) calloc(blksize, sizeof(int));
   tc = (float *) calloc(blksize, sizeof(float));
   tr = (float *) calloc(blksize, sizeof(float));

   for (i=0; i < blksize; i++)                  /* init piv matrix */
      *(piv+i) = i;

   LUFact(Mat, piv, blksize);                   /* perform LU factorization */

/*
```

```
 To find inverse, repeatedly solve LU system for columns from the
 identity matrix, and place results into the inverse matrix.
*/

  for (i=0; i < blksize; i++) {
     for (j=0; j < blksize; j++)                    /* Set-up identity matrix */
        *(tc+j) = 0.0;
     *(tc+i) = 1.0;
     LUSolv(Mat, piv, blksize, tc, tr);             /* Now solve */
     for (j=0; j < blksize; j++)                    /* Copy results into Matinv */
        *(Matinv+i*blksize+j) = *(tr+j);
     }

  free(piv);
  free(tc);
  free(tr);

} /* Invert */

/*
  MatMult - Multiplies two matricies, producing a third.

            C = A x B

            A - m x n
            B - n x p
            C - m x p
*/

void MatMult(A,B,C,m,n,p)

float *A;                    /* A is m x n */
float *B;                    /* B is n x p */
float *C;                    /* C is m x p */
int m,n,p;                   /* matrix dimensions */

{
  int i,j,k;                /* Loop counters */
  float sum, t1, t2;        /* temporary storage */

  for (i=0; i < m; i++)
     for (j=0; j < p; j++) {
        sum =0.0;
        for (k=0; k < n; k++) {
           t1 = *(A+i*n+k);
           t2 = *(B+k*p+j);
           if (t1 && t2)
              sum += t1 * t2;
           }
        if (fabs(sum) > Zero)
           *(C+i*p+j) = sum;
        else
           *(C+i*p+j) = 0.0;
        }
} /* MatMult */

/*
  VectMult - Performs the following operation:

             c = A x b
```

```
            Where A is m x n, b is n x 1, and c is m x 1.
*/

void VectMult(A,b,c,m,n)

float *A;                    /* A is m x n */
float b[];                   /* b is n x 1 */
float c[];                   /* c is m x 1 */
int m,n;                     /* matrix dimensions */

{
  int i,j;                  /* loop counters */
  float sum, t1, t2;        /* temporary storage */

  for (i=0; i < m; i++) {
     sum = 0.0;
     for (j=0; j < n; j++) {
        t1 = *(A+i*n+j);
        t2 = *(b+j);
        if (t1 && t2)
           sum += t1 * t2;
        }
     if (fabs(sum) > Zero)
        *(c+i) = sum;
     else
        *(c+i) = 0.0;
     }

} /* VectMult */

/*

  DotProd - given two vectors of equal length, this procedure calculates
            the dot product of them.

            DotProd = a . b

*/

float DotProd(a,b,m)

float a[];              /* the first vector */
float b[];              /* the second vector */
int m;                  /* the length of the vectors a & b */

{

  int i;               /* loop counter */
  float t1, t2, sum; /* temporary storage */

  sum = 0.0;
  for (i=0; i < m; i++) {
      t1 = *(a+i);
      t2 = *(b+i);
      if (t1 && t2)
         sum += t1 * t2;
      }

  if (fabs(sum) < Zero)
```

```
    sum = 0.0;

  return(sum);

} /* DotProd */


/*
  FindWorkingSet - find all equations whose constraints are currently
                   valid. Accepts as input: A, b, x.  Those rows of
                   A for which Ax=b are placed in the working set.
                   Those rows which
*/

void FindWorkingSet(A,b,x,W,L,m,n,q,r)

float *A;                /* A is an m x n constraints matrix */
float b[];               /* b is an m x 1 constraingin values matrix */
float x[];               /* x constains the current location */
int W[];                 /* W is a vector of indicies stating which rows */
                        /*   are in the working set */
int L[];                 /* N is a vector of inidicies of those rows of A */
                        /* which are not in the working set */
int m,n;                 /* matrix dimensions */
int *q;                  /* the size of the working set */
int *r;                  /* the size of the non-working set */

{
  int i;                /* Counter */
  float *temp;          /* used to compare calculated b with actual b */

  void VectMult();
  char *calloc();

  temp = (float *) calloc(m, sizeof(float));

  VectMult(A,x,temp,m,n);  /* Get calculated values of b */

  *q = 0;
  *r = 0;
  for (i=0; i < m; i++)
     if ( fabs( *(temp+i) - *(b+i) ) < Zero )
        W[(*q)++] = i;
     else
        L[(*r)++] = i;

  free(temp);

} /* FindWorkingSet */

/*
  AddToSet - add an element to the specified set.
*/

void AddToSet(val,S,size)

int val;                 /* the value to add to the set S */
int S[];                 /* the set the val will be added to */
int *size;               /* the current size of S */
```

```
{
  int i,j,k;              /* counters */

  j = *size;
  k = val;

  for (i=0; i < *size; i++)
     if (k < S[i]) {
        j = i;
        k  = S[*size];
        }

  for (i=*size; i > j; i--)
     S[i] = S[i-1];

  S[j] = val;

  (*size)++;

} /* AddToSet */



/*
  Deactivate - a procedure that removes the input value from the
               working set. If the value is not in the working set,
               no changes are made to the working set.
*/

void Deactivate(val,W,L,q,r)

int val;                  /* the equation to be deleted from the working set */
int W[];                  /* the working set */
int L[];                  /* the non-working set */
int *q;                   /* *q is the number of elements in the working set */
int *r;                   /* *r is the number of elements in the lazy set */

{
  int i,j;                /* counters */

  void AddToSet();

  j = 0;
  for (i=0; i < *q; i++)
     if (val != W[i])
        W[j++] = W[i];

  AddToSet(val,L,r);

  (*q)--;

} /* Deactivate */


/*
  FindAlpha1 - A function that returns a calculated value for Alpha1.
               This function calculates Alpha1 as follows:

                          b - Ax
          Alpha1 = (min) --------    for all constraints not in the
```

```
                                    x'QPQx
                  Alpha2Cand = ----------
                                   x'QPQPQx

*/

float FindAlpha2Cand(x,Q,P,PQ,n)

float x[];          /* x is the current position in the constraint space */
float *Q;           /* Q is the coefficients of the function to be minimized */
float *P;           /* P is the projection matrix */
float *PQ;          /* PQ is the result (previously calculated) of P x Q */
int n;              /* x is n x 1; P, Q are n x n. */

{
  float num, den;           /* Temp. storage for numerator & denominator */
  float *QPQ;               /* Storage for the result of Q x P x Q */
  float *QPQPQ;             /* Storage for the result of QPQPQ */
  float *temp4;             /* Temporary storage vector */
  float A2Cand;             /* Alpha2Candidate temporary storage */

  void MatMult();
  void VectMult();
  float DotProd();
  char *calloc();

  QPQ = (float *) calloc(n*n, sizeof(float));
  QPQPQ = (float *) calloc(n*n, sizeof(float));
  temp4 = (float *) calloc(n, sizeof(float));

  MatMult(Q,PQ,QPQ,n,n,n);  /* Calculate Q x P x Q */

  VectMult(QPQ,x,temp4,n,n);      /* temp4 = QPQx */
  num = DotProd(x,temp4,n);       /* num = x'QPQx */

  MatMult(QPQ,PQ,QPQPQ,n,n,n);    /* temp3 = QPQPQ */
  VectMult(QPQPQ,x,temp4,n,n);    /* temp4 = QPQPQx */
  den = DotProd(x,temp4,n);       /* den = x'QPQPQx */

  if (den < Zero)
     A2Cand = 1E30;                  /* or any large number will do */
  else
     A2Cand = num/den;

  free(QPQ);
  free(QPQPQ);
  free(temp4);

  return(A2Cand);

} /* FindAlpha2Cand */

/*
  MakeIdent - make an n x n identity matrix
*/

void MakeIdent(I,n)

float *I;                  /* the identity matrix */
int n;                     /* the size of the identity matrix */
```

```
                        A * D       working set

*/

float FindAlpha1(A,b,x,D,L,m,n,r)

float *A;                /* A is an m x n constraints matrix */
float b[];               /* b is an m x 1 constraining values matrix */
float x[];               /* x contains the current location in constraint space*
float D[];               /* D constains the current optimal direction vector *,
int L[];                 /* L is a matrix of indicies indicating which rows */
                        /*   in A form the non-working set (the lazy set). */
int m,n;                 /* matrix dimensions */
int r;                   /* The # of elements in the non-working set */

{
  int i,j;              /* loop counters */
  float num,t1,t2;      /* temporary variables for calc. numerator */
  float den,dt2;        /* temporary variables for calc. denominator */
  float comp;
  float min;

  /* calculate comparison values and check to see if they are the smallest */

  for (i=0; i < r; i++) {
     num = b[L[i]];
     den = 0.0;
     for (j=0; j < n; j++) {          /* calculate comparison values */
        t1 = *(A+L[i]*n+j);
        t2 = x[j];
        dt2 = D[j];
        if (t1 && t2)
           num -= t1 * t2;
        if (t1 && dt2)
           den += t1 * dt2;
        }
     if ( fabs(den) > Zero )          /* compare with minimum */
        comp = num / den;
     else
        comp = min + 2;               /* ignore den == 0 cases */
     if (!i)
        min = fabs(comp) + 1;
     if (comp > Zero) {
        if (comp < min)
           min = comp;
        }
     }

  return(min);

} /* FindAlpha1 */


/*
  FindAlpha2Cand - this function calculates and returns a value for
                   Alpha2Candidate. This value is then compared with
                   Alpha1 to see if it or Alpha1 will become Alpha2.

                   Alpha2Candidate is calculated as follows:
```

```
{
  int i,j;                /* loop counters */

  for (i=0; i < n; i++)
     for (j=0; j < n; j++)
         *(I+i*n+j) = (i == j) ? 1.0 : 0.0;

} /* MakeIdent */

/*
   AqAqInverse - this procedure calculates the value of:  inv(Aq x Aq'),
                 where the inv() operation is the matrix inverse.

                 This procedure is called with a flag that can contain
                 up to three values:

                 0 - perform a straight inverse
                 1 - perform an inverse update after an operation to add
                     to the working set
                 2 - perform an inverse update after an operation to
                     delete from the working set

*/

void AqAqInverse(AqAqInv,A,W,m,n,q,Flag)

float *AqAqInv;                 /* This will hold the result inv(Aq x Aq') */
float *A;                       /* Constraint equations matrix             */
int W[];                        /* the working set */
int m,n;                        /* the size of A is m x n */
int q;                          /* the size of the working set */
int Flag;                       /* a Flag, as described above */

{
  int i,j,k;                  /* loop counters */
  float *temp1;               /* temporary result of Aq x Aq' */
  float sum, t1, t2;          /* temporary storage elements */

  char *calloc();

if (Flag == 0) {

  temp1 = (float *) calloc(q*q, sizeof(float));

  /* find Aq x Aq' */

  for (i=0; i < q; i++)
     for (j=0; j < q; j++) {
        sum = 0.0;
        for (k=0; k < n; k++) {
           t1 = *(A+W[i]*n+k);
           t2 = *(A+W[j]*n+k);
           if (t1 && t2)
              sum += t1 * t2;
           }
     *(temp1+i*q+j) = sum;
     }

  Invert(temp1,AqAqInv,q);
```

```
  free(temp1);
  }

} /* AqAqInverse */

/*

  CalcP - a procedure to calculate the value of the projection matrix, P.
          P is found from the following equation:

                      P = I - Aq' x inv(Aq x Aq') x Aq
*/

void CalcP(P,A,I,AqAqInv,W,m,n,q)

float *P;                          /* Projection matrix */
float *A;                          /* Constraint equations matrix */
float *I;                          /* the n x n identity matrix */
float *AqAqInv;                    /* this holds the result of inv(Aq x Aq') */
int W[];                           /* the working set */
int m,n;                           /* A matrix is of size m x n */
int q;                             /* the size of the working set */

{
  int i,j,k;                      /* loop counters */
  float *temp1;                   /* used to hold the result of AqAqInv x Aq */
  float sum, t1, t2;              /* temporary storage */

  char *calloc();

  temp1 = (float *) calloc(q*n, sizeof(float));

  for (i=0; i < q; i++)
     for (j=0; j < n; j++) {
        sum = 0.0;
        for (k=0; k < q; k++) {
           t1 = *(AqAqInv+i*q+k);
           t2 = *(A+W[k]*n+j);
           if (t1 && t2)
              sum += t1 * t2;
           }
        *(temp1+i*n+j) = sum;       /* temp1 = AqAqInv x Aq */
        }

  for (i=0; i < n; i++)
      for (j=0; j < n; j++) {
         sum = *(I+i*n+j);
         for (k=0; k < q; k++) {
             t1 = *(A+W[k]*n+i);
             t2 = *(temp1+k*n+j);
             if (t1 && t2)
                sum -= t1 * t2;
             }
         *(P+i*n+j) = sum;     /* P = I - Aq' x AqAqInv x Aq */
         }

  free(temp1);

} /* CalcP */
```

```
/*
  CalcD - A procedure to calculate the D vector. D is n x 1, and represents
          the direction of greatest decrease for the objective function.
          D is calculated from the following equation:

                                D = - PQx
*/

void CalcD(D,P,Q,PQ,x,n)

float D[];                  /* D is the direction of greatest descent */
float *P;                   /* P is the Projection Matrix */
float *Q;                   /* Q is the coefficients of the objective fn. */
float *PQ;                  /* PQ is used to return the value P x Q */
float x[];                  /* x is the current position in constraint space */
int n;                      /* n is the number of variables */

{
  int i;                   /* loop counters */

  void MatMult();
  void VectMult();

  MatMult(P,Q,PQ,n,n,n);    /* PQ = P x Q */

  VectMult(PQ,x,D,n,n);     /* D = PQx */

  for (i=0; i < n; i++)
      *(D+i) = - *(D+i);    /* D = -D */

} /* CalcD */

/*
  CalcLagrange - a procedure to calculate Lagrange multipliers.
                 Returns a n x 1 vector of Langrange multipliers,
                 calculated from the following equation:

                    Lambda = - inv(AqAq')AqQx
*/

void CalcLagrange(Lambda, A, AqAqInv, Q, x, W, m, n, q)

float Lambda[];              /* n x 1 vector of Lagrange multipliers */
float *A;                    /* constraint equations matrix */
float *AqAqInv;              /* AqAqInv = inv(Aq x Aq') */
float *Q;                    /* objective function coefficients */
float x[];                   /* current position in constraint space */
int W[];                     /* the current Working set */
int m;                       /* the # of constraint equations */
int n;                       /* the number fo variables (size of x) */
int q;                       /* the size of the working set */

{
  int i,j;                  /* loop counters */
  float *Qx;                /* temporary storage for the result Qx */
  float *AqQx;              /* temporary.storage for the result AqQx */
  float sum, t1, t2;        /* temporary storage */

  void VectMult();
```

```
  char *calloc();

  Qx = (float *) calloc(n, sizeof(float));
  AqQx = (float *) calloc(n, sizeof(float));

  VectMult(Q,x,Qx,n,n);       /* Qx = Q * x */

  /* now find AqQx */

  for (i=0; i < q; i++) {
     sum = 0.0;
     for (j=0; j < n; j++) {
        t1 = *(A+W[i]*n+j);
        t2 = *(Qx+j);
        if (t1 && t2)
           sum += t1 * t2;
        }
     *(AqQx+i) = sum;
     }

  /* now calculate Lambda */

  VectMult(AqAqInv,AqQx,Lambda,q,q);     /* Lambda = AqAqInv * Aq * Q * x  */

  /* now find negative */

 for (i=0; i < q; i++)
    *(Lambda+i) = - *(Lambda+i);

 free(Qx);
 free(AqQx);

} /* CalcLagrange */

/*
  EvalLambda - a procedure to evaluate the vector of Lagrangians.
               If the vector is greater than or equal to -Zero, then
               the pos flag is set to 1, otherwise it is set to 0.
               If the pos flag is 0, the minimum negative value of
               the Lagrangian vector will be found. MinPos returns the
               position of the most negative Lagrangian.
*/

void EvalLambda(Lambda, n, pos, MinPos)

float Lambda[];                 /* the vector of Lagrangians */
int n;                          /* the size of the vector of Lagrangians */
int *pos;                       /* if Lambda >= -Zero, pos is set to 1 */
int *MinPos;                    /* the location of the most neg. Lagr. */

{
  int i;                        /* Loop counter */
  float min;                    /* used t0 find MinPos */

  min = 0.0;
  *pos = 1;
  *MinPos = -1;

  for (i=0; i < n; i++)
     if (Lambda[i] < -Zero) {
```

```
        *pos = 0;
        if (Lambda[i] < min) {
           min = Lambda[i];
           *MinPos = i;
           }
        }

} /* EvalLambda */


/*

  CalcObj - Calculate the value of the objective function, x'Qx

*/

float CalcObj(Q,x,n)

float *Q;                       /* coefficients of the objective function */
float x[];                      /* current position in the constraint space */
int n;                          /* Q is n x n, x is n x 1 */

{
  float *Qx;              /* Used to hold results of Qx */
  float result;           /* temporary storage */

  void VectMult();
  char *calloc();

  Qx = (float *) calloc(n, sizeof(float));

  VectMult(Q,x,Qx,n,n);         /* Qx = Q * x */

  result = DotProd(x,Qx,n);

  free(Qx);

  return(result);

} /* CalcObj */

/*
  Update_x - a procedure to update the x variable:  x = x + Alpha2 * D
*/

void Update_x(x,Alpha2,D,n)

float x[];                      /* position in constraint space */
float Alpha2;                   /* the amount to move in direction D */
float D[];                      /* the direction of increased optimality */
int n;                          /* the size of x and D */

{
  int i;                       /* counter */

  for (i=0; i < n; i++)
     x[i] += Alpha2 * D[i];

} /* Update_x */
```

```
/*
  PrintLambda - a procedure to print the Langrangian vector, and the
                information EvalLambda calculated about this vector.
                For debug or analysis purposes only.
*/

void PrintLambda(Lambda,n,pos,MinPos)

float Lambda[];              /* the Lagrangian vector */
int n;                       /* the length of Lambda */
int pos;                     /* is Lambda all positive. A 1 so indicates */
int MinPos;                  /* the position of the most neg. element */

{
  int i;                    /* loop counter */

  printf("\n\nLagrangian vector:\n\n[");
  for (i=0; i < n; i++) {
     if (i) printf(", ");
     printf("%6.3f",Lambda[i]);
     }
  printf(" ]\n\n");

  printf("Is Lambda positive: %s\n\n", (pos == 1) ? "Yes" : "No");

  printf("Most negative element is #%2d\n\n",MinPos);

} /* PrintLambda */



/*
  PrintStatus - This routine prints the current state of the Minimize
                procedure. Use this procedure for debugging or analysis
                only (the printing takes a lot of time).
*/

void PrintStatus(A,b,Q,x,W,L,P,D,iter,m,n,q,r)

float *A;                    /* constraint equations matrix */
float b[];                   /* constraining values matrix */
float *Q;                    /* objective function coefficients matrix */
float x[];                   /* position in constraint space */
int W[];                     /* the current working set */
int L[];                     /* the current non-working set */
float *P;                    /* the projection matrix */
float D[];                   /* the direction of optimality */
int *iter;                   /* the current iteration number */
int m;                       /* total number of constraints */
int n;                       /* the number of variables */
int q;                       /* the size of the working set */
int r;                       /* the size of the non-working set */

{
  int i,j;

  float CalcObj();

  printf("\n\n\n==========================================================\n\n");
  printf("        Iteration #%3d\n\n",*iter);
```

```
printf("Working Set:\n\n[");
for (i=0; i < q; i++) {
   if (i) printf(", ");
   printf("%2d",W[i]);
   }
printf(" ]\n\n");

printf("Working Set of Equations:\n\n");
for (i=0; i < q; i++) {
   printf("[");
   for (j=0; j < n; j++) {
      if (j) printf(", ");
      printf("%4.1f",*(A+W[i]*n+j));
      }
   printf(" : %4.1f ]\n",b[W[i]]);
   }

printf("\nNon-Working Set:\n\n[");
for (i=0; i < r; i++) {
   if (i) printf(", ");
   printf("%2d",L[i]);
   }
printf(" ]\n\n");

printf("Non-Working Set of Equations:\n\n");
for (i=0; i < r; i++) {
   printf("[");
   for (j=0; j < n; j++) {
      if (j) printf(", ");
      printf("%4.1f",*(A+L[i]*n+j));
      }
   printf(" : %4.1f ]\n",b[L[i]]);
   }

printf("\nCurrent position:\n\n[");
for (i=0; i < n; i++) {
   if (i) printf(", ");
   printf("%6.3f",x[i]);
   }
printf(" ]\n\n");

printf("Current Projection Matrix:\n\n");
for (i=0; i < n; i++) {
   printf("[ ");
   for (j=0; j < n; j++) {
      if (j) printf(", ");
      printf("%6.3f",*(P+i*n+j));
      }
   printf(" ]\n");
   }

printf("\n\nCurrent Direction Matrix:\n\n[");
for (i=0; i < n; i++) {
   if (i) printf(", ");
   printf("%6.3f",D[i]);
   }
printf(" ]\n\n");

printf("Current Value of Objective Function: %6.3f\n\n", CalcObj(Q,x,n));
```

```
} /* PrintStatus */


/*
   Minimize - a procedure to minimize the function x'Qx subject to
              linear constraints (A Quadratic Programming Solver).
              Uses a modified Gradient Projection Method to accomplish
              the task of minimization.

     Input:
             A - an m x n array of constraint equations
             b - a m x 1 vector of constraining values
             Q - the Q in x'Qx (cost function coefficients) (n x n)
             x - an initial feasible point (n x 1)
             m - the number of equations in A
             n - the number of variables in x

    Output:
             mc - the minimum value of the objective function
             iter - the number of iterations it took to find mc
             x - the final optimum position
*/

void Minimize(A,b,Q,x,m,n,mc,iter,Debug)

float *A;                    /* constraint equations matrix */
float b[];                   /* constraining values vector */
float *Q;                    /* objective function coefficients matrix */
float x[];                   /* position in constraint space */
int m,n;                     /* array sizes */
float *mc;                   /* value of objective function at minimum */
int *iter;                   /* the number of iterations performed */
int Debug;                   /* if set to 1, will print debugging info */

{
  int i,j;                  /* loop counters */
  float Alpha1;             /* used to calculate Alpha2 */
  float Alpha2Cand;         /* used to calculate Alpha2 */
  float Alpha2;             /* used to update x vector */
  float *P;                 /* the Projection Matrix */
  float *D;                 /* the direction of optimality */
  float *AqAqInv;           /* used to store the value of inv(Aq * Aq') */
  float *PQ;                /* Used to store the value of P * Q  */
  float *Lambda;            /* will contain the value of the Lagrangians */
  float *I;                 /* Used to constain the n x n identity matrix */
  int *W;                   /* used to store the working set */
  int *L;                   /* used to store the non-working set */
  int q;                    /* the size of the working set */
  int r;                    /* the size of the non-working set */
  int Flag;                 /* used to denote whether an equation was */
                            /* added, deleted, or nothing happened to the */
                            /* working set (1,2 and 0 respectively) */
  int D_Equal_0;            /* A flag. Set to 1 if D = 0, set to 0 otherwise
  int pos;                  /* A flag used to evaluate Lambda >= 0 */
  int MinPos;               /* used to hold the most negative pos. in Lambda */

  void FindWorkingSet();
```

```
void Deactivate();
float FindAlpha1();
float FindAlpha2Cand();
void MakeIdent();
void AqAqInverse();
void CalcP();
void CalcD();
void CalcLagrange();
void Update_x();
void EvalLambda();
float CalcObj();
void PrintStatus();
char *calloc();


/* initialize variables */

*iter = 0;
*mc = 0.0;
q = 0;
r = 0;
Flag = 0;

P = (float *) calloc(n*n, sizeof(float));
D = (float *) calloc(n, sizeof(float));
AqAqInv = (float *) calloc(m*m, sizeof(float));
PQ = (float *) calloc(n*n, sizeof(float));
Lambda = (float *) calloc(m, sizeof(float));
I = (float *) calloc(n*n, sizeof(float));
W = (int *) calloc(m, sizeof(int));
L = (int *) calloc(m, sizeof(int));

MakeIdent(I,n);

Loop_A:

FindWorkingSet(A,b,x,W,L,m,n,&q,&r);

Loop_B:

AqAqInverse(AqAqInv,A,W,m,n,q,0);
CalcP(P,A,I,AqAqInv,W,m,n,q);

Loop_C:

CalcD(D,P,Q,PQ,x,n);

(*iter)++;

if (Debug)
   PrintStatus(A,b,Q,x,W,L,P,D,iter,m,n,q,r);

D_Equal_0 = 1;
for (i=0; i < n; i++)
   if (fabs(D[i]) > Zero)
      D_Equal_0 = 0;

if (D_Equal_0 == 0) {

   Alpha1 = FindAlpha1(A,b,x,D,L,m,n,r);
```

```
      Alpha2Cand = FindAlpha2Cand(x,Q,P,PQ,n);

      if (Alpha2Cand <= Alpha1)
         Alpha2 = Alpha2Cand;
      else
         Alpha2 = Alpha1;

      if (Debug)
         printf("\n\nAlpha1: %6.3f\nAlpha2Cand: %6.3f\nAlpha2: %6.3f",Alpha1,
                Alpha2Cand,Alpha2);

      Update_x(x,Alpha2,D,n);

      if (Alpha1 == Alpha2) {
         Flag = 1;
         goto Loop_A;
         }
      else {
         Flag = 0;
         goto Loop_C;
         }
      }

   else {

      CalcLagrange(Lambda,A,AqAqInv,Q,x,W,m,n,q);

      EvalLambda(Lambda,n,&pos,&MinPos);

      if (Debug)
         PrintLambda(Lambda,n,pos,MinPos);

      if (pos == 1)
         goto Optimized;

      Deactivate(W[MinPos],W,L,&q,&r);

      goto Loop_B;
      }

   Optimized:

   *mc = CalcObj(Q,x,n);

   free(P);
   free(D);
   free(AqAqInv);
   free(PQ);
   free(Lambda);
   free(I);
   free(W);
   free(L);

} /* Minimize */


/*

   main  - accept input from the user, then call the routine Minimize.
           Once Minimize has finished, print the final results and quit.
```

```
*/

main()

/*
  What follows is a sample variable declaration block for using the
  gradient projection algorithm.
*/

{
  float *A;                    /* Constraint matrix                          */
  float *b;                    /* Constraint values matrix                   */
  float *Q;                    /* Costs matrix                               */
  float *x;                    /* Objective variable values                  */
  float mc;                    /* Used to hold the minimum cost at opt.      */
  int  M, N;                   /* Used to specify size of A actually used    */
  int  iter;                   /* Used to find number of iterations          */
  int Debug;                   /* Set this to one for debugging info.        */


  char *VarName;               /* Used to name variables in x vector         */
  int i,j;                     /* Loop counters                              */

  char *calloc();

/* Start reading input from the user */

  printf("\n\nQuadratic Program Solver\n");
  printf("(Gradient Projection Method)\n\n\n");

  M = Mmax;
  N = Nmax;

  while (N >= Nmax) {
     printf("Please enter the number of variables: ");
     scanf("%d",&N);
     if (N > Nmax)
        printf("\nPlease enter a number in the range (1 - %2d)\n",Nmax);
     }

  while (M >= Mmax) {
     printf("\nPlease enter the number of constraint equations: ");
     scanf("%d",&M);
     if (M > Mmax)
        printf("\nPlease enter a number in the range (1 - %2d)\n",Mmax);
     }

  /*
     Now allocate space for all the arrays
  */

  A = (float *) calloc(M*N, sizeof(float));
  b = (float *) calloc(M, sizeof(float));
  Q = (float *) calloc(N*N, sizeof(float));
  x = (float *) calloc(N, sizeof(float));
  VarName = (char *) calloc(N*10, sizeof(char));


  printf("\n\n\nPlease enter a ten character (or less) description of ");
```

```
printf("each variable.\n");

for (i=0; i < N; i++) {
   printf("\nVariable %2d: ",(i+1));
   scanf("%10s",VarName+10*i);
   }

printf("\n\n\nFor each constraint equation, please enter all ");
printf("variable coefficients below:\n");

for (i=0; i < M; i++) {
   printf("\nEquation #%2d: \n\n",(i+1));
   for (j=0; j < N; j++) {
      printf("Coefficient of %s : ",VarName+10*j);
      scanf("%f",A+i*N+j);
      }
   printf("\nConstraining value: ");
   scanf("%f",b+i);
   }

printf("\n\nPlease enter an initial starting point for x from which\n");
printf("minimization will proceed.\n\n");

for (i=0; i < N; i++) {
    printf("\n Intial value of %s : ",VarName+10*i);
    scanf("%f",x+i);
    }

printf("\n\nPlease enter a 1 for debugging information, 0 for none: ");
scanf("%d",&Debug);

printf("\n\n");

/* Hardwire in the values for Q to minimize the euclidian norm */

*(Q+0)  = 1.0;
*(Q+1)  = 0.0;
*(Q+2)  = -1.0;
*(Q+3)  = 0.0;

*(Q+4)  = 0.0;
*(Q+5)  = 1.0;
*(Q+6)  = 0.0;
*(Q+7)  = -1.0;

*(Q+8)  = -1.0;
*(Q+9)  = 0.0;
*(Q+10)  = 1.0;
*(Q+11)  = 0.0;

*(Q+12)  = 0.0;
*(Q+13)  = -1.0;
*(Q+14)  = 0.0;
*(Q+15)  = 1.0;

/* Input complete */

Minimize(A,b,Q,x,M,N,&mc,&iter,Debug);

printf("\n\nFinal variable values:\n\n");
```

```
   for (i=0; i < N; i++)
      printf("\n%s : %10.5f\n",VarName+10*i,*(x+i));

   printf("\nMinimum value of objective function: %10.5f\n",mc);

   printf("\nNumber of iterations performed: %d\n",iter);

   free(A);
   free(b);
   free(Q);
   free(x);
   free(VarName);

   printf("\n\nDone.\n");
}
```

PROGRAM LISTING

To produce an object file from this source code, enter:

On a Sun 3:

cc gpm.c -f68881 -lm

On a Sun 4:

cc gpm.c -lm

```
#include <stdio.h>
#include <math.h>

#define Mmax 10
#define Nmax 10
#define Zero 0.0000001

/*

   Quadratic Programming Problem Solver
   ------------------------------------

     By: Jim Whitehead
   Date: May 13, 1989


   This program is to be implemented as a procedure for inclusion in
   a larger program. (Will be written as a complete program for testing
   purposes.)

   This program will solve a quadratic programming problem by use of the
   gradient projection method. The program solves a quadratic programming
   problem expressed in the following form:

        minimize     x'Cx

        subject to   Ax = b

   In this representation, C is a quadratic coeffiecients matrix, A is a
   constraint matrix, b are the constraining values, and x is the variable
   to be optimized.

*/

/*

  LUFact - A procedure that accepts as input an array, and provides
           as output two arrays, L & U, such that LU = A (the array).
           A pivot vector is an input (& output). This vectors is used
           to perform pivot operations without any copying of matrix
           elements. Pivoting two rows is performed by swapping indicies
           in the pivot matrix.
*/

int LUFact(array, piv, blksize)

float array[][Nmax];
int piv[];
int blksize;

{
   int i, j, k, p, tp, tpiv;
   double max, temp;

   /* following algorithm on p. 345 of Burden & Faires: */

   /* Step 1 */
   max = array[0][0];
   tp = 0;
   for (p = 1; p < blksize ; p++) {
```

```
    if (fabs(array[p][0]) > max) {
       tp = p;
       max = fabs(array[p][0]);
    }
}
if (max == 0.0) {
   printf("LUFact: Pivot error (max is zero)");
   exit(1);
}
/* Step 2 */
if (tp != 0) {
   *piv = tp;
   *(piv+tp) = 0;
}
/* Step 4 */
for (j = 1; j < blksize; j++) {
    array[*(piv+j)][0] = (array[*(piv+j)][0] / array[*piv][0]);
}
/* Step 5 */
for (i=1; i < (blksize-1); i++) {
    /* Step 6 */
    max = 0.0;
    tp = i;
    for (j=i; j < blksize; j++) {
        temp = array[*(piv+j)][i];
        k = 0;
        do {
           temp = temp - array[*(piv+j)][k] * array[*(piv+k)][i];
           }
        while (k++ < (i-1));
        temp = fabs(temp);
        if (temp > max) {
           max = temp;
           tp = j;
        }
    }
    /* Step 7 */
    if (tp != i) {
       tpiv = *(piv+tp);
       *(piv+tp) = *(piv+i);
       *(piv+i)  = tpiv;
    }
    /* Step 8 */
    temp = 0.0;
    k = 0;
    do {
       temp = temp + array[*(piv+i)][k] * array[*(piv+k)][i]; /* + OK*/
       }
    while ( k++ < (i-1));
    array[*(piv+i)][i] -= temp;

    /* Step 9 */
    for (j = i+1; j < blksize ; j++) {
        temp = array[*(piv+i)][j];
        k = 0;
        do {
           temp = temp - array[*(piv+i)][k] * array[*(piv+k)][j];
           }
        while (k++ < (i-1));
        array[*(piv+i)][j] = temp;
```

```
            temp = array[*(piv+j)][i];
            k = 0;
            do {
               temp = temp - array[*(piv+j)][k] * array[*(piv+k)][i];
               }
            while ( k++ < (i-1));
            array[*(piv+j)][i] = temp / array[*(piv+i)][i];
         }
      }
      /* Step 10 */
      temp = array[*(piv+blksize-1)][blksize-1];
      for (k=0; k < (blksize-1); k++)
          temp = temp-array[*(piv+blksize-1)][k] * array[*(piv+k)][blksize-1];
      if (temp == 0.0 ) {
          printf("LUFact: Step 10 -- No unique solution exists.");
          exit(1);
      }
      array[*(piv+blksize-1)][blksize-1] = temp;
} /* End lufact */

/*
 LUSolv - a function to solve the system of equations: LUx = b,
          where L and U are lower and upper triangular matricies
          and x and b are column vectors.

 Input:
        Mat - a block matrix containing L and U stored in compact form
              (i.e., not storing diagonals of L matrix)
        piv - a pivot value matrix (from LUFact routine)
        blksize - the size of Mat
        b - the b in LUx = b
        x - the x in LUx = b (function output)
*/

void LUSolv(Mat, piv, blksize, b, x)

float Mat[][Nmax];   /* Holds LU factored matrix */
int piv[];           /* pivot values matrix      */
int blksize;         /* tells how big Mat is     */
float b[];           /* from LUx = b             */
float x[];           /* from LUx = b (output)    */

{
  float z[Mmax];         /* Used for intermediate value storage */
  int i,j;              /* loop counters */
  float dec;             /* temp. storage variable */

  z[0] = b[0];

  for (i=1; i < blksize; i++) {
     dec = *(b+i);
     for (j=0; j <= (i-1); j++)
        dec -= Mat[*(piv+i)][j] * *(z+j);
     if ( fabs(dec) < Zero )
        dec = 0.0;
     *(z+i) = dec;
     }

  x[blksize-1] = z[blksize-1] / Mat[*(piv+blksize-1)][blksize-1];
```

```
  for (i=blksize-2; i>= 0 ; i--) {
     dec = *(z+i);
     for (j=i+1; j<blksize; j++)
        dec -= Mat[*(piv+i)][j] * *(x+j);
     if ( *((*(Mat+i))+i) )
        dec = dec / Mat[*(piv+i)][i];
     if ( fabs(dec) < Zero )
        dec = 0.0;
     *(x+i) = dec;
     }
} /* LUSolv */


/*

 Invert - a function to calculate the inverse of a block matrix

 Input:
         Mat - matrix to be inverted
         Matinv - the inverted matrix
         blksize - the size of each matrix

*/

void Invert(Mat, Matinv, blksize)

float Mat[][Nmax];       /* the matrix to be inverted */
float Matinv[][Mmax];    /* the inverted matrix       */
int blksize;             /* the size of the matricies */

{
  int piv[Mmax];    /* a pivot matrix used in the LUfact routine */
  float tc[Mmax];   /* a temporary vector used in calc. of inverse */
  float tr[Mmax];   /* a temporary results vector used in calcs. */
  int i, j;

  int LUFact();
  void LUSolv();

  for (i=0; i < blksize; i++)                    /* init piv matrix */
     *(piv+i) = i;

  LUFact(Mat, piv, blksize);                     /* perform LU factorization */

/*
 To find inverse, repeatedly solve LU system for columns from the
 identity matrix, and place results into the inverse matrix.
*/

  for (i=0; i < blksize; i++) {
     for (j=0; j < blksize; j++)                 /* Set-up identity matrix */
        *(tc+j) = 0.0;
     *(tc+i) = 1.0;
     LUSolv(Mat, piv, blksize, tc, tr);          /* Now solve */
     for (j=0; j < blksize; j++)                 /* Copy results into Matinv
        Matinv[i][j] = *(tr+j);
     }
} /* Invert */
```

```
/*
  MatMult - Multiplies two matricies, producing a third.

            C = A x B

            A - m x n
            B - n x p
            C - m x p
*/

void MatMult(A,B,C,m,n,p)

float A[][Mmax];            /* A is m x n */
float B[][Mmax];            /* B is n x p */
float C[][Mmax];            /* C is m x p */
int m,n,p;                  /* matrix dimensions */

{
  int i,j,k;                /* Loop counters */
  float sum, t1, t2;        /* temporary storage */

  for (i=0; i < m; i++)
     for (j=0; j < p; j++) {
        sum =0.0;
        for (k=0; k < n; k++) {
           t1 = A[i][k];
           t2 = B[k][j];
           if (t1 && t2)
              sum += t1 * t2;
           }
        if (fabs(sum) > Zero)
           C[i][j] = sum;
        else
           C[i][j] = 0.0;
        }
} /* MatMult */

/*
  VectMult - Performs the following operation:

             c = A x b

             Where A is m x n, b is n x 1, and c is m x 1.
*/

void VectMult(A,b,c,m,n)

float A[][Mmax];            /* A is m x n */
float b[];                  /* b is n x 1 */
float c[];                  /* c is m x 1 */
int m,n;                    /* matrix dimensions */

{
  int i,j;                  /* loop counters */
  float sum, t1, t2;        /* temporary storage */

  for (i=0; i < m; i++) {
     sum = 0.0;
     for (j=0; j < n; j++) {
        t1 = A[i][j];
```

```
        t2 = b[j];
        if (t1 && t2)
           sum += t1 * t2;
        }
     if (fabs(sum) > Zero)
        c[i] = sum;
     else
        c[i] = 0.0;
     }

} /* VectMult */

/*

  DotProd - given two vectors of equal length, this procedure calculates
            the dot product of them.

            DotProd = a . b

*/

float DotProd(a,b,m)

float a[];              /* the first vector */
float b[];              /* the second vector */
int m;                  /* the length of the vectors a & b */

{

  int i;               /* loop counter */
  float t1, t2, sum; /* temporary storage */

  sum = 0.0;
  for (i=0; i < m; i++) {
      t1 = *(a+i);
      t2 = *(b+i);
      if (t1 && t2)
         sum += t1 * t2;
      }

  if (fabs(sum) < Zero)
     sum = 0.0;

  return(sum);

} /* DotProd */


/*
  FindWorkingSet - find all equations whose constraints are currently
                   valid. Accepts as input: A, b, x.  Those rows of
                   A for which Ax=b are placed in the working set.
                   Those rows which
*/

void FindWorkingSet(A,b,x,W,L,m,n,q,r)

float A[][Nmax];          /* A is an m x n constraints matrix */
float b[];                /* b is an m x 1 constraingin values matrix */
float x[];                /* x constains the current location */
```

```
int W[];                    /* W is a vector of indicies stating which rows */
                           /*   are in the working set */
int L[];                    /* N is a vector of inidicies of those rows of A */
                           /* which are not in the working set */
int m,n;                    /* matrix dimensions */
int *q;                     /* the size of the working set */
int *r;                     /* the size of the non-working set */

{
  int i;                   /* Counter */
  float temp[Mmax];        /* used to compare calculated b with actual b */

  void VectMult();


  VectMult(A,x,temp,m,n);  /* Get calculated values of b */

  *q = 0;
  *r = 0;
  for (i=0; i < m; i++)
     if ( fabs(temp[i] - b[i]) < Zero )
        W[(*q)++] = i;
     else
        L[(*r)++] = i;

} /* FindWorkingSet */

/*
  AddToSet - add an element to the specified set.
*/

void AddToSet(val,S,size)

int val;                    /* the value to add to the set S */
int S[];                    /* the set the val will be added to */
int *size;                  /* the current size of S */

{
  int i,j,k;               /* counters */

  j = *size;
  k = val;

  for (i=0; i < *size; i++)
     if (k < S[i]) {
        j = i;
        k  = S[*size];
        }

  for (i=*size; i > j; i--)
     S[i] = S[i-1];

  S[j] = val;

  (*size)++;

  /* AddToSet */
```

```
/*
  Deactivate - a procedure that removes the input value from the
               working set. If the value is not in the working set,
               no changes are made to the working set.
*/

void Deactivate(val,W,L,q,r)

int val;                    /* the equation to be deleted from the working set */
int W[];                    /* the working set */
int L[];                    /* the non-working set */
int *q;                     /* *q is the number of elements in the working set */
int *r;                     /* *r is the number of elements in the lazy set */

{
  int i,j;                 /* counters */

  void AddToSet();

  j = 0;
  for (i=0; i < *q; i++)
     if (val != W[i])
        W[j++] = W[i];

  AddToSet(val,L,r);

  (*q)--;

} /* Deactivate */


/*
  FindAlpha1 - A function that returns a calculated value for Alpha1.
               This function calculates Alpha1 as follows:

                          b - Ax
          Alpha1 = (min) --------    for all constraints not in the
                          A * D       working set

*/

float FindAlpha1(A,b,x,D,L,n,r)

float A[][Nmax];          /* A is an m x n constraints matrix */
float b[];                /* b is an m x 1 constraining values matrix */
float x[];                /* x contains the current location in constraint space*
float D[];                /* D constains the current optimal direction vector */
int L[];                  /* L is a matrix of indicies indicating which rows */
                         /*   in A form the non-working set (the lazy set). */
int n;                  /* matrix dimensions */
int r;                    /* The # of elements in the non-working set */

{
  int i,j;               /* loop counters */
  float num,t1,t2;       /* temporary variables for calc. numerator */
  float den,dt2;         /* temporary variables for calc. denominator */
  float comp;
  float min;

  /* calculate comparison values and check to see if they are the smallest */
```

```
  for (i=0; i < r; i++) {
     num = b[L[i]];
     den = 0.0;
     for (j=0; j < n; j++) {          /* calculate comparison values */
        t1 = A[L[i]][j];
        t2 = x[j];
        dt2 = D[j];
        if (t1 && t2)
           num -= t1 * t2;
        if (t1 && dt2)
           den += t1 * dt2;
        }
     if ( fabs(den) > Zero )          /* compare with minimum */
        comp = num / den;
     else
        comp = min + 2;               /* ignore den == 0 cases */
     if (!i)
        min = fabs(comp) + 1;
     if (comp > Zero) {
        if (comp < min)
           min = comp;
        }
     }

  return(min);

} /* FindAlpha1 */


/*
  FindAlpha2Cand - this function calculates and returns a value for
                   Alpha2Candidate. This value is then compared with
                   Alpha1 to see if it or Alpha1 will become Alpha2.

                   Alpha2Candidate is calculated as follows:

                                        x'QPQx
                   Alpha2Cand = ----------
                                       x'QPQPQx

*/

float FindAlpha2Cand(x,Q,P,PQ,n)

float x[];          /* x is the current position in the constraint space */
float Q[][Nmax];    /* Q is the coefficients of the function to be minimized */
float P[][Nmax];    /* P is the projection matrix */
float PQ[][Nmax];   /* PQ is the result (previously calculated) of P x Q */
int n;              /* x is n x 1; P, Q are n x n. */

{
  float num, den;             /* Temp. storage for numerator & denominator */
  float QPQ[Nmax][Nmax];      /* Storage for the result of Q x P x Q */
  float QPQPQ[Nmax][Nmax];    /* Storage for the result of QPQPQ */
  float temp4[Nmax];          /* Temporary storage vector */
  float A2Cand;               /* Alpha2Candidate temporary storage */

  void MatMult();
  void VectMult();
```

```
  float DotProd();

  MatMult(Q,PQ,QPQ,n,n,n);  /* Calculate Q x P x Q */

  VectMult(QPQ,x,temp4,n,n);      /* temp4 = QPQx */
  num = DotProd(x,temp4,n);       /* num = x'QPQx */

  MatMult(QPQ,PQ,QPQPQ,n,n,n);    /* temp3 = QPQPQ */
  VectMult(QPQPQ,x,temp4,n,n);    /* temp4 = QPQPQx */
  den = DotProd(x,temp4,n);       /* den = x'QPQPQx */

  if (den < Zero)
     A2Cand = 1000;                    /* or any large number will do */
  else
     A2Cand = num/den;

  return(A2Cand);

} /* FindAlpha2Cand */

/*
  MakeIdent - make an n x n identity matrix
*/

void MakeIdent(I,n)

float I[][Nmax];           /* the identity matrix */
int n;                     /* the size of the identity matrix */

{
  int i,j;                /* loop counters */

  for (i=0; i < n; i++)
     for (j=0; j < n; j++)
         I[i][j] = (i == j) ? 1.0 : 0.0;

} /* MakeIdent */

/*
   AqAqInverse - this procedure calculates the value of:  inv(Aq x Aq'),
                 where the inv() operation is the matrix inverse.

                 This procedure is called with a flag that can contain
                 up to three values:

                 0 - perform a straight inverse
                 1 - perform an inverse update after an operation to add
                     to the working set
                 2 - perform an inverse update after an operation to
                     delete from the working set

*/

void AqAqInverse(AqAqInv,A,W,n,q,Flag)

float AqAqInv[][Mmax];          /* This will hold the result inv(Aq x Aq') */
float A[][Nmax];                /* Constraint equations matrix */
int W[];                        /* the working set */
int n;                          /* the number of varaibles */
int q;                          /* the size of the working set */
```

```
int Flag;                          /* a Flag, as described above */

{
  int i,j,k;                      /* loop counters */
  float temp1[Mmax][Mmax];        /* temporary result of Aq x Aq' */
  float sum, t1, t2;              /* temporary storage elements */


if (Flag == 0) {

  /* find Aq x Aq' */

  for (i=0; i < q; i++)
     for (j=0; j < q; j++) {
        sum = 0.0;
        for (k=0; k < n; k++) {
           t1 = A[W[i]][k];
           t2 = A[W[j]][k];
           if (t1 && t2)
              sum += t1 * t2;
           }
     temp1[i][j] = sum;
     }

  Invert(temp1,AqAqInv,q);
  }

} /* AqAqInverse */

/*

  CalcP - a procedure to calculate the value of the projection matrix, P.
          P is found from the following equation:

                      P = I - Aq' x inv(Aq x Aq') x Aq
*/

void CalcP(P,A,I,AqAqInv,W,n,q)

float P[][Nmax];                   /* Projection matrix */
float A[][Nmax];                   /* Constraint equations matrix */
float I[][Nmax];                   /* the n x n identity matrix */
float AqAqInv[][Mmax];             /* this holds the result of inv(Aq x Aq') */
int W[];                           /* the working set */
int n;                             /* the number of variables */
int q;                             /* the size of the working set */

{
  int i,j,k;                      /* loop counters */
  float temp1[Mmax][Nmax];        /* used to hold the result of AqAqInv x Aq */
  float temp2[Nmax][Nmax];        /* holds result of Aq' x AqAqInv x Aq */
  float sum, t1, t2;              /* temporary storage */

  for (i=0; i < q; i++)
     for (j=0; j < n; j++) {
        sum = 0.0;
        for (k=0; k < q; k++) {
           t1 = AqAqInv[i][k];
           t2 = A[W[k]][j];
           if (t1 && t2)
```

```
                sum += t1 * t2;
            }
          temp1[i][j] = sum;        /* temp1 = AqAqInv x Aq */
          }

   for (i=0; i < n; i++)
       for (j=0; j < n; j++) {
          sum = I[i][j];
          for (k=0; k < q; k++) {
              t1 = A[W[k]][i];
              t2 = temp1[k][j];
              if (t1 && t2)
                 sum -= t1 * t2;
              }
          P[i][j] = sum;      /* P = I - Aq' x AqAqInv x Aq */
          }

} /* CalcP */

/*
  CalcD - A procedure to calculate the D vector. D is n x 1, and represents
          the direction of greatest decrease for the objective function.
          D is calculated from the following equation:

                                D = - PQx
*/

void CalcD(D,P,Q,PQ,x,n)

float D[];                  /* D is the direction of greatest descent */
float P[][Nmax];            /* P is the Projection Matrix */
float Q[][Nmax];            /* Q is the coefficients of the objective fn. */
float PQ[][Nmax];           /* PQ is used to return the value P x Q */
float x[];                  /* x is the current position in constraint space */
int n;                      /* n is the number of variables */

{
  int i;                   /* loop counters */

  void MatMult();
  void VectMult();

  MatMult(P,Q,PQ,n,n,n);    /* PQ = P x Q */

  VectMult(PQ,x,D,n,n);     /* D = PQx */

  for (i=0; i < n; i++)
      *(D+i) = - *(D+i);    /* D = -D */

} /* CalcD */

/*
  CalcLagrange - a procedure to calculate Lagrange multipliers.
                 Returns a n x 1 vector of Langrange multipliers,
                 calculated from the following equation:

                    Lambda = - inv(AqAq')AqQx
*/

void CalcLagrange(Lambda, A, AqAqInv, Q, x, W, n, q)
```

```
float Lambda[];                 /* n x 1 vector of Lagrange multipliers */
float A[][Nmax];                /* constraint equations matrix */
float AqAqInv[][Nmax];          /* AqAqInv = inv(Aq x Aq') */
float Q[][Nmax];                /* objective function coefficients */
float x[];                      /* current position in constraint space */
int W[];                        /* the current Working set */
int n;                          /* the number fo variables (size of x) */
int q;                          /* the size of the working set */

{
  int i,j;                     /* loop counters */
  float Qx[Nmax];              /* temporary storage for the result Qx */
  float AqQx[Nmax];            /* temporary storage for the result AqQx */
  float sum, t1, t2;           /* temporary storage */

  void VectMult();

  VectMult(Q,x,Qx,n,n);         /* Qx = Q * x */

  /* now find AqQx */

  for (i=0; i < q; i++) {
     sum = 0.0;
     for (j=0; j < n; j++) {
        t1 = A[W[i]][j];
        t2 = Qx[j];
        if (t1 && t2)
           sum += t1 * t2;
        }
     AqQx[i] = sum;
     }

  /* now calculate Lambda */

  VectMult(AqAqInv,AqQx,Lambda,q,q);     /* Lambda = AqAqInv * Aq * Q * x  */

  /* now find negative */

 for (i=0; i < q; i++)
    *(Lambda+i) = - *(Lambda+i);

} /* CalcLagrange */

/*
  EvalLambda - a procedure to evaluate the vector of Lagrangians.
               If the vector is greater than or equal to -Zero, then
               the pos flag is set to 1, otherwise it is set to 0.
               If the pos flag is 0, the minimum negative value of
               the Lagrangian vector will be found. MinPos returns the
               position of the most negative Lagrangian.
*/

void EvalLambda(Lambda, n, pos, MinPos)

float Lambda[];                 /* the vector of Lagrangians */
int n;                          /* the size of the vector of Lagrangians */
int *pos;                       /* if Lambda >= -Zero, pos is set to 1 */
int *MinPos;                    /* the location of the most neg. Lagr. */
```

```
{
  int i;                        /* Loop counter */
  float min;                    /* used t0 find MinPos */

  min = 0.0;
  *pos = 1;
  *MinPos = -1;

  for (i=0; i < n; i++)
     if (Lambda[i] < -Zero) {
        *pos = 0;
        if (Lambda[i] < min) {
           min = Lambda[i];
           *MinPos = i;
           }
        }

} /* EvalLambda */


/*

  CalcObj - Calculate the value of the objective function, x'Qx

*/

float CalcObj(Q,x,n)

float Q[][Nmax];                /* coefficients of the objective function */
float x[];                      /* current position in the constraint space */
int n;                          /* Q is n x n, x is n x 1 */

{
  float Qx[Nmax];               /* Used to hold results of Qx */

  void VectMult();


  VectMult(Q,x,Qx,n,n);         /* Qx = Q * x */

  return(DotProd(x,Qx,n));

} /* CalcObj */

/*
  Update_x - a procedure to update the x variable:  x = x + Alpha2 * D
*/

void Update_x(x,Alpha2,D,n)

float x[];                      /* position in constraint space */
float Alpha2;                   /* the amount to move in direction D */
float D[];                      /* the direction of increased optimality */
int n;                          /* the size of x and D */

{
  int i;                        /* counter */

  for (i=0; i < n; i++)
     x[i] += Alpha2 * D[i];
```

```
} /* Update_x */

/*
  PrintLambda - a procedure to print the Langrangian vector, and the
                information EvalLambda calculated about this vector.
                For debug or analysis purposes only.
*/

void PrintLambda(Lambda,n,pos,MinPos)

float Lambda[];              /* the Lagrangian vector */
int n;                       /* the length of Lambda */
int pos;                     /* is Lambda all positive. A 1 so indicates */
int MinPos;                  /* the position of the most neg. element */

{
  int i;                    /* loop counter */

  printf("\n\nLagrangian vector:\n\n[");
  for (i=0; i < n; i++) {
     if (i) printf(", ");
     printf("%6.3f",Lambda[i]);
     }
  printf(" ]\n\n");

  printf("Is Lambda positive: %s\n\n", (pos == 1) ? "Yes" : "No");

  printf("Most negative element is #%2d\n\n",MinPos);

  /* PrintLambda */



/*
  PrintStatus - This routine prints the current state of the Minimize
                procedure. Use this procedure for debugging or analysis
                only (the printing takes a lot of time).
*/

void PrintStatus(A,b,Q,x,W,L,P,D,iter,n,q,r)

float A[][Nmax];             /* constraint equations matrix */
float b[];                   /* constraining values matrix */
float Q[][Nmax];             /* objective function coefficients matrix */
float x[];                   /* position in constraint space */
int W[];                     /* the current working set */
int L[];                     /* the current non-working set */
float P[][Nmax];             /* the projection matrix */
float D[];                   /* the direction of optimality */
int *iter;                   /* the current iteration number */
int n;                       /* the number of variables */
int q;                       /* the size of the working set */
int r;                       /* the size of the non-working set */

{
  int i,j;

  float CalcObj();
```

```
printf("\n\n\n============================================================\n\n");
printf("        Iteration #%3d\n\n",*iter);

printf("Working Set:\n\n[");
for (i=0; i < q; i++) {
   if (i) printf(", ");
   printf("%2d",W[i]);
   }
printf(" ]\n\n");

printf("Working Set of Equations:\n\n");
for (i=0; i < q; i++) {
   printf("[");
   for (j=0; j < n; j++) {
      if (j) printf(", ");
      printf("%4.1f",A[W[i]][j]);
      }
   printf(" : %4.1f ]\n",b[W[i]]);
   }

printf("\nNon-Working Set:\n\n[");
for (i=0; i < r; i++) {
   if (i) printf(", ");
   printf("%2d",L[i]);
   }
printf(" ]\n\n");

printf("Non-Working Set of Equations:\n\n");
for (i=0; i < r; i++) {
   printf("[");
   for (j=0; j < n; j++) {
      if (j) printf(", ");
      printf("%4.1f",A[L[i]][j]);
      }
   printf(" : %4.1f ]\n",b[L[i]]);
   }

printf("\nCurrent position:\n\n[");
for (i=0; i < n; i++) {
   if (i) printf(", ");
   printf("%6.3f",x[i]);
   }
printf(" ]\n\n");

printf("Current Projection Matrix:\n\n");
for (i=0; i < n; i++) {
   printf("[ ");
   for (j=0; j < n; j++) {
      if (j) printf(", ");
      printf("%6.3f",P[i][j]);
      }
   printf(" ]\n");
   }

printf("\n\nCurrent Direction Matrix:\n\n[");
for (i=0; i < n; i++) {
   if (i) printf(", ");
   printf("%6.3f",D[i]);
   }
printf(" ]\n\n");
```

```
  printf("Current Value of Objective Function: %6.3f\n\n", CalcObj(Q,x,n));

} /* PrintStatus */


/*
   Minimize - a procedure to minimize the function x'Qx subject to
              linear constraints (A Quadratic Programming Solver).
              Uses a modified Gradient Projection Method to accomplish
              the task of minimization.

     Input:
             A - an m x n array of constraint equations
             b - a m x 1 vector of constraining values
             Q - the Q in x'Qx (cost function coefficients) (n x n)
             x - an initial feasible point (n x 1)
             m - the number of equations in A
             n - the number of variables in x

    Output:
             mc - the minimum value of the objective function
             iter - the number of iterations it took to find mc
             x - the final optimum position
*/

void Minimize(A,b,Q,x,m,n,mc,iter,Debug)

float A[][Nmax];              /* constraint equations matrix */
float b[];                    /* constraining values vector */
float Q[][Nmax];              /* objective function coefficients matrix */
float x[];                    /* position in constraint space */
int m,n;                      /* array sizes */
float *mc;                    /* value of objective function at minimum */
int *iter;                    /* the number of iterations performed */
int Debug;                    /* if set to 1, will print debugging info */

{
  int i,j;                   /* loop counters */
  float Alpha1;              /* used to calculate Alpha2 */
  float Alpha2Cand;          /* used to calculate Alpha2 */
  float Alpha2;              /* used to update x vector */
  float P[Nmax][Nmax];       /* the Projection Matrix */
  float D[Nmax];             /* the direction of optimality */
  float AqAqInv[Mmax][Mmax]; /* used to store the value of inv(Aq * Aq') */
  float PQ[Nmax][Nmax];      /* Used to store the value of P * Q  */
  float Lambda[Mmax];        /* will contain the value of the Lagrangians */
  float I[Nmax][Nmax];       /* Used to constain the n x n identity matrix */
  int W[Mmax];               /* used to store the working set */
  int L[Mmax];               /* used to store the non-working set */
  int q;                     /* the size of the working set */
  int r;                     /* the size of the non-working set */
  int Flag;                  /* used to denote whether an equation was */
                             /* added, deleted, or nothing happened to the */
                             /* working set (1,2 and 0 respectively) */
  int D_Equal_0;             /* A flag. Set to 1 if D = 0, set to 0 otherwise */
  int pos;                   /* A flag used to evaluate Lambda >= 0 */
  int MinPos;                /* used to hold the most negative pos. in Lambda */
```

```
void FindWorkingSet();
void Deactivate();
float FindAlpha1();
float FindAlpha2Cand();
void MakeIdent();
void AqAqInverse();
void CalcP();
void CalcD();
void CalcLagrange();
void Update_x();
void EvalLambda();
float CalcObj();
void PrintStatus();


/* initialize variables */

*iter = 0;
*mc = 0.0;
q = 0;
r = 0;
Flag = 0;

MakeIdent(I,n);

Loop_A:

FindWorkingSet(A,b,x,W,L,m,n,&q,&r);

Loop_B:

AqAqInverse(AqAqInv,A,W,n,q,0);
CalcP(P,A,I,AqAqInv,W,n,q);

Loop_C:

CalcD(D,P,Q,PQ,x,n);

(*iter)++;

if (Debug)
   PrintStatus(A,b,Q,x,W,L,P,D,iter,n,q,r);

D_Equal_0 = 1;
for (i=0; i < n; i++)
   if (fabs(D[i]) > Zero)
      D_Equal_0 = 0;

if (D_Equal_0 == 0) {

   Alpha1 = FindAlpha1(A,b,x,D,L,n,r);
   Alpha2Cand = FindAlpha2Cand(x,Q,P,PQ,n);

   if (Alpha2Cand <= Alpha1)
      Alpha2 = Alpha2Cand;
   else
      Alpha2 = Alpha1;

   if (Debug)
```

```
        printf("\n\nAlpha1: %6.3f\nAlpha2Cand: %6.3f\nAlpha2: %6.3f",Alpha1,
               Alpha2Cand,Alpha2);

     Update_x(x,Alpha2,D,n);

     if (Alpha1 == Alpha2) {
        Flag = 1;
        goto Loop_A;
        }
     else {
        Flag = 0;
        goto Loop_C;
        }
     }

  else {

     CalcLagrange(Lambda,A,AqAqInv,Q,x,W,n,q);

     EvalLambda(Lambda,n,&pos,&MinPos);

     if (Debug)
        PrintLambda(Lambda,n,pos,MinPos);

     if (pos == 1)
        goto Optimized;

     Deactivate(W[MinPos],W,L,&q,&r);

     goto Loop_B;
     }

  Optimized:

  *mc = CalcObj(Q,x,n);

} /* Minimize */


/*

  main  - accept input from the user, then call the routine Minimize.
          Once Minimize has finished, print the final results and quit.

*/

main()

/*
  What follows is a sample variable declaration block for using the
  gradient projection algorithm.
*/

{
  float A[Mmax][Nmax];        /* Constraint matrix                          */
  float b[Mmax];              /* Constraint values matrix                   */
  float Q[Nmax][Nmax];        /* Costs matrix                               */
  float x[Nmax];              /* Objective variable values                  */
  float mc;                   /* Used to hold the minimum cost at opt.      */
  int  M, N;                  /* Used to specify size of A actually used    */
```

```
  int  iter;                  /* Used to find number of iterations       */
  int Debug;                  /* Set this to one for debugging info.     */


  char VarName[Mmax][10];     /* Used to name variables in x vector      */
  int i,j;                    /* Loop counters                           */

/* Start reading input from the user */

  printf("\n\nQuadratic Program Solver\n");
  printf("(Gradient Projection Method)\n\n\n");

  M = Mmax;
  N = Nmax;

  while (N >= Nmax) {
     printf("Please enter the number of variables: ");
     scanf("%d",&N);
     if (N > Nmax)
        printf("\nPlease enter a number in the range (1 - %2d)\n",Nmax);
     }

  while (M >= Mmax) {
     printf("\nPlease enter the number of constraint equations: ");
     scanf("%d",&M);
     if (M > Mmax)
        printf("\nPlease enter a number in the range (1 - %2d)\n",Mmax);
     }

  printf("\n\n\nPlease enter a ten character (or less) description of ");
  printf("each variable.\n");

  for (i=0; i < N; i++) {
     printf("\nVariable %2d: ",(i+1));
     scanf("%10s",VarName[i]);
     }

  printf("\n\n\nFor each constraint equation, please enter all ");
  printf("variable coefficients below:\n");

  for (i=0; i < M; i++) {
     printf("\nEquation #%2d: \n\n",(i+1));
     for (j=0; j < N; j++) {
        printf("Coefficient of %s : ",VarName[j]);
        scanf("%f",&A[i][j]);
        }
     printf("\nConstraining value: ");
     scanf("%f",&b[i]);
     }

  printf("\n\nPlease enter an initial starting point for x from which\n");
  printf("minimization will proceed.\n\n");

  for (i=0; i < N; i++) {
      printf("\n Intial value of %s : ",VarName[i]);
      scanf("%f",&x[i]);
      }

  printf("\n\nPlease enter a 1 for debugging information, 0 for none: ");
  scanf("%d",&Debug);
```

```
   printf("\n\n");

   /* Hardwire in the values for Q to minimize the euclidian norm */

  Q[0][0] = 1.0;
  Q[0][1] = 0.0;
  Q[0][2] = -1.0;
  Q[0][3] = 0.0;

  Q[1][0] = 0.0;
  Q[1][1] = 1.0;
  Q[1][2] = 0.0;
  Q[1][3] = -1.0;

  Q[2][0] = -1.0;
  Q[2][1] = 0.0;
  Q[2][2] = 1.0;
  Q[2][3] = 0.0;

  Q[3][0] = 0.0;
  Q[3][1] = -1.0;
  Q[3][2] = 0.0;
  Q[3][3] = 1.0;

   /* Input complete */

   Minimize(A,b,Q,x,M,N,&mc,&iter,Debug);

   printf("\n\nFinal variable values:\n\n");

   for (i=0; i < N; i++)
      printf("\n%s : %10.5f\n",VarName[i],x[i]);

   printf("\nMinimum value of objective function: %10.5f\n",mc);

   printf("\nNumber of iterations performed: %d\n",iter);

   printf("\n\nDone.\n");
}
```

REFERENCES

Linear Programming:

Dantzig, George B., Linear Programming and Extensions, 1963, Princeton University Press.

Solow, Daniel, Linear Programming: An Introduction to Finite Improvement Algorithms, North Holland, 1984.

Quadratic Programming:

Kyriakopoulos, Kostas, An Efficient Minimum Distance and Collision Estimation Technique for On-Line Motion Planning of Robotic Manipulation, CIRSSE Report #19, 1989.

Luenberger, David G., Linear and Nonlinear Programming, Second Edition, Addison-Wesley, 1984.

LU Factorization:

Burden, Richard L. and Douglas J. Faires, Numerical Analysis, Third Edition, Prindle, Weber & Schmidt, 1985.